Trabajo de fin de grado

Grado en Ingeniería Informática

Análisis de la factibilidad de la aplicación de técnicas de Deep Learning para el procesamiento de lenguaje natural a la generación de secuencias de optimizaciones en la compilación de programas SW

Autor: Jesús Morales Millán

Cádiz, Octubre, 2020

Trabajo de fin de grado

Grado en Ingeniería Informática

Análisis de la factibilidad de la aplicación de técnicas de Deep Learning para el procesamiento de lenguaje natural a la generación de secuencias de optimizaciones en la compilación de programas SW

Autor: Jesús Morales Millán
Director: Bernabé Dorronsoro Díaz
Subdirector: Juan Carlos de la Torre Macías

Cádiz, Octubre, 2020

Contenido

Índice de figuras

1. Introducción

Hoy en día, debido al exponencial crecimiento de la tecnología tanto en su uso como en su evolución, han surgido multitud de nuevos lenguajes de programación. Como consecuencia, es necesario desarrollar nuevos compiladores. Cada uno hace una implementación diferente, de modo que todos consiguen como resultado un código máquina ejecutable por un computador, pero con distinto nivel de optimización.

A partir de este hecho, nacen compiladores que traducen lenguajes de alto nivel en un lenguaje de nivel intermedio para su posterior tratamiento. De este modo, todos ellos consiguen el mismo nivel de optimización al ser tratados como un mismo lenguaje. En estos, tras múltiples pruebas, se han desarrollado unas pautas de optimización general denominadas *optimizaciones* o *passes*.

Optimizar un código informático consiste entonces en la aplicación de una secuencia de passes que tienen como resultado un código ejecutable que ofrece mejor rendimiento que el código fuente original. Esta secuencia de passes tiene una longitud y una relación de orden indeterminadas. La búsqueda de la secuencia óptima ha sido y es fuente de investigación por parte de la comunidad científica, la cual ha sido abarcada desde múltiples enfoques, entre ellos [1]:

- Problema de selección: se enfoca en qué passes se encuentran en la secuencia, independientemente de su orden.
- Problema de ordenación de fases: en el que se forma una secuencia de optimizaciones en un determinado orden. Para ello, se forman grupos de passes teniendo en cuenta el impacto de rendimiento que tienen unos sobre otros. Se tiene en cuenta, además, los resultados del anterior problema (cuáles deben usarse). Aún no se ha resuelto, consiguiendo únicamente avanzar en el estudio del impacto.

La unidad mínima de un lenguaje con información es la palabra y, de una secuencia de optimización, el pass. Asimismo, una frase es un conjunto de palabras con un sentido completo. Nuestra idea es que somos capaces de crear *frases* a partir de otras. Es decir, subyace que una secuencia de passes que obtengan una optimización de calidad a partir del código fuente, existe un lenguaje que la rige. Si existe un lenguaje oculto en la secuencia de passes, debería ser posible abarcar el problema de la secuencia de passes como si de un problema de lenguaje natural (NLP por sus siglas en inglés) se tratase.

El presente trabajo modela el conocido problema de la secuencialización de optimizaciones utilizando técnicas propias de los problemas NLP.

1.1. Motivación

Debido a que se utiliza un mismo lenguaje de nivel intermedio para todos los de alto nivel, el lenguaje de optimización es el mismo para todos, como se ha mencionado previamente. De este modo, en vez de establecer unas pautas generales para cualquier programa, podemos buscar la optimización específica de cada programa mediante el estudio como lenguaje natural. Conseguiríamos así un nivel de optimización que puede mejorar el nivel actualmente establecido.

2. Introducción teórica

2.1. LLVM

A día de hoy, como se comentaba previamente, debido al exponencial crecimiento de la tecnología, han surgido multitud de nuevos lenguajes de programación. Como consecuencia, es necesario desarrollar nuevos compiladores. Estos, desde un enfoque clásico, están formados por tres elementos (figura 1) [7]:

- Un frontend, que analiza y busca errores en el código fuente para finalmente construir un árbol general que representa la sintaxis del programa, llamado AST (Abstract Syntax Tree). Este puede ser convertido de nuevo a otra representación antes del siguiente paso.
- Un optimizador, que busca reducir el tiempo de ejecución eliminando secciones de código redundantes o innecesarias.
- Por último, se genera el código usando un determinado juego de instrucciones, aprovechando las ventajas que ofrece la arquitectura usada.

Figura 1: Compilador con estructura previamente descrita.[7]

De esta forma, para cada lenguaje se ha de escribir un nuevo compilador, y para arquitectura un nuevo generador de código. Es aquí donde entra en juego LLVM, el cual se diferencia de este modelo clásico, proporcionándonos ventajas.

Cuando mencionamos el nombre LLVM, se puede estar haciendo alusión a varios elementos distintos. De todos ellos, el más usado y el que engloba a los demás, es el llamado proyecto LLVM.

Este un proyecto formado por otros a su vez, tales como compiladores y toolbox, que, en su totalidad, forman un compilador completo. Está formado por tres partes: un front end, una etapa de optimización/transformaciones (donde se busca la optimización), y un backend. [19]

A continuación se detalla las ventajas de LLVM y el funcionamiento de cada una de sus partes (además de resumida gráficamente en la figura 2).

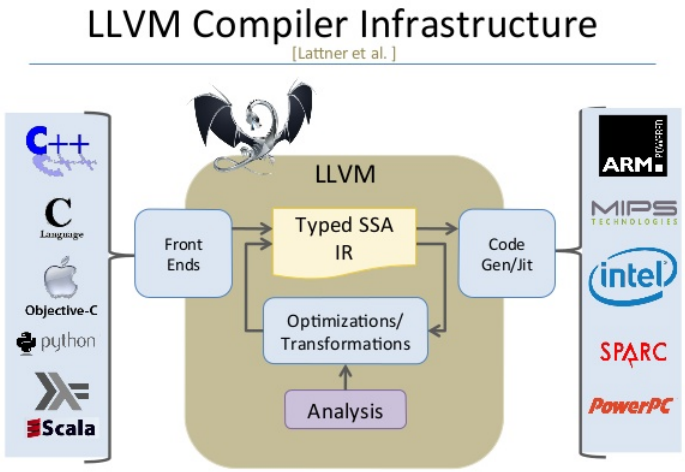


Figura 2: Infraestructura de LLVM donde podemos ver todas sus partes. [3]

2.1.1. ¿Qué nos ofrece LLVM?

Las ventajas del proyecto LLVM frente a un compilador clásico son múltiples, entre ellas encontramos:

- Nos ofrece la conversión de algoritmos de múltiples lenguajes de alto nivel a múltiples arquitecturas de computadores.
- Debido a la separación de este en tres diferenciadas partes, para que LLVM funcione con un nuevo lenguaje, solo se tendría que implementar un nuevo Frontend. Del mismo modo, para una nueva arquitectura únicamente sería necesario un Backend acorde a esta.
- Todos los lenguajes son tratados de la misma manera de forma interna, luego todos tienen acceso al mismo nivel de optimización.
- El proyecto permite la inclusión de nuevas optimizaciones de código, además de documentación de cómo implementarlo de manera sencilla mediante el mecanismo de herencia.
- Considerable mejora en tiempo de ejecución y compilación frente a otros compiladores (figura 3).

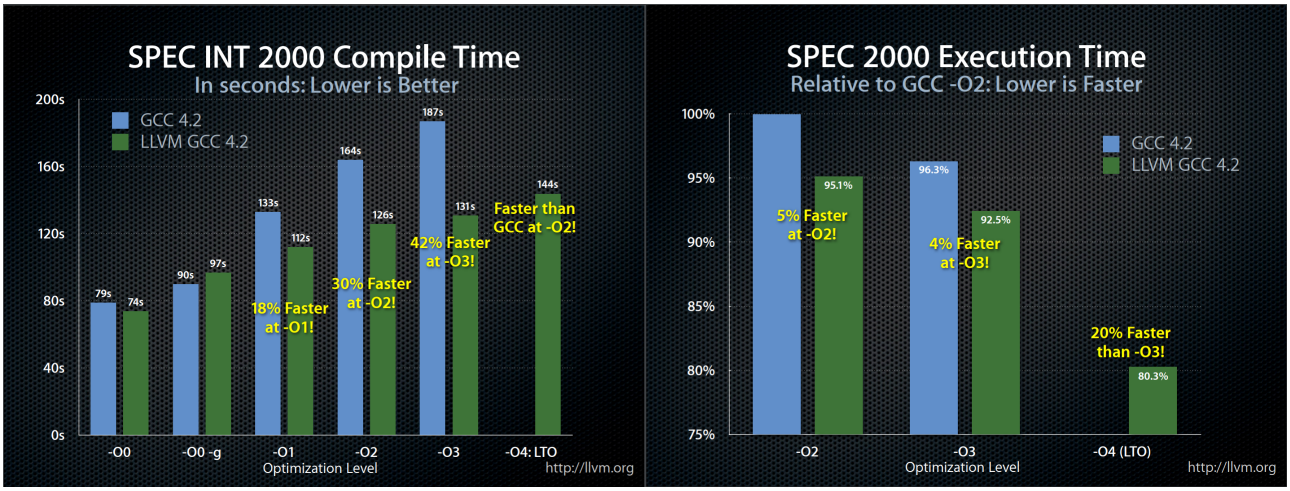


Figura 3: Comparativa entre GCC y LLVM. [15]

2.1.2. Frontend

El "Frontend" de LLVM es el encargado de traducir lenguajes de programación de alto nivel, tales como C, C++, Objective-C, Objective-C++, Python, Perl, Scala, entre otros, en un lenguaje de nivel intermedio entre alto nivel y ensamblador propio de LLVM llamado "LLVM IR".

Todos los lenguajes de alto nivel se traducen a este, de forma que **para cada nuevo lenguaje solo se tiene que construir un nuevo Frontend**, sin afectar al optimizador ni al generador.

Esta labor, en el caso de los cuatros primeros lenguajes mencionados (C, C++, Objective-C, Objective-C++), se lleva a cabo mediante Clang, uno de los subproyectos de LLVM, el cual tiene como objetivo la reducción del tiempo de compilación y el uso de mensajes de error muy explicativos, haciendo así que sea amigable de cara al programador. Además, debido a su licencia de software libre tipo BDS, permite que sea utilizado con uso comercial.

Los pasos a realizar en este Frontend son [8]:

- Preprocesado: Se procesan directivas tal y como la inclusión de paquetes, definiciones de macros, directinas condicionales, etc.
- Análisis léxico (scanner): También conocido como tokenización, cada palabra del código se clasifica en categorías o tokens, aplicando unos patrones previamente definidos. Por ejemplo, si nos encontramos que una palabra cumple con el patrón "int|float|char", nos encontraríamos ante el token/categoría "typedef" (definición de tipo). Generalmente, se suelen usar cinco categorías de tokens (figura 4):
- Análisis sintáctico (parser): En esta frase, se comprueba si los tokens obtenidos son válidos, es decir, pertenecen a la gramática del lenguaje. Esta comprobación se realiza usando reglas recursivas. Un ejemplo podría ser:

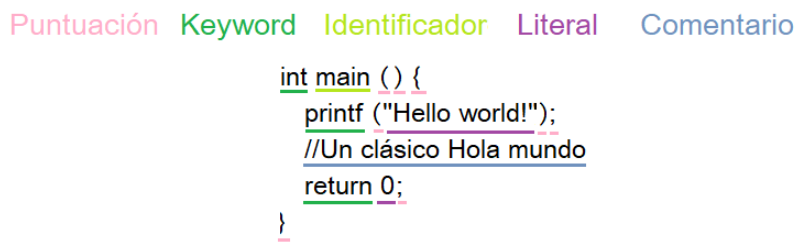


Figura 4: Ejemplo de tokenización

```
instruccion: KEYWORD IDENTIFICADOR              Regla inicial->instruccion
instruccion: IDENTIFICADOR = expresion
expresion: expresion + expresion
expresion: expresion - expresion
expresion: expresion * expresion
expresion: expresion / expresion
expresion: IDENTIFICADOR
```

Las instrucciones “int a” y “a=a+b+c+d” serían tratadas como correctas dada la gramática del lenguaje definida por esas reglas.

- Análisis semántico (type checking): Se comprueba que el significado del código dado sea válido en busca de errores. Puede que la expresión “c=a+b” sea correcta según nuestra gramática, pero si c no ha sido declarado, o si a y b son booleanos cuya operación suma no ha sido definida, tenemos errores.
- Generación de código IR.

Como apunte final del Frontend, es necesario resaltar que una vez realizada la traducción, el algoritmo de lenguaje LLVM IR es codificado en un formato propio de LLVM llamado Bitcode (“.bc”). Este está formado por dos elementos:

- Un conjunto de elementos llamados "bits", que a pesar de su nombre, no son bits y se encargan de describir línea a línea el contenido del algoritmo.
- La propia codificación del algoritmo IR, con la que decodificar los bits a código.

2.1.3. Transformaciones: Análisis y optimización

Tras obtener el código IR, se aplican transformaciones al código. Estas transformaciones han sido implementadas como subclases de una clase definida llamada Pass, de forma que a estas transformaciones de LLVM se les suele conocer como “passes” [20]. Los passes se aplican con distintos fines, entre ellos [16]:

- Recolectar información con fines de depuración o visualización del programa. Estos son llamados de **análisis**.
- Realizar mutaciones en el código. Estos son los llamados de **transformación**.
- Realizar acciones útiles que no se pueden categorizar en las dos anteriores. Por ejemplo, hay una transformación, -transform-warning, para emitir un mensaje de aviso (warning) de qué transformaciones no han sido aplicadas hasta ese momento. Estos son los llamados de **utilidad**.

Estas transformaciones son aplicadas por **opt**, la herramienta de optimización de LLVM.

El objetivo principal es, mediante el análisis y las mutaciones, optimizar el código de forma que este sea más eficiente.

Es importante destacar que las mutaciones realizadas **pueden** optimizar el tiempo de ejecución. Si se aplican los passes indicados en el orden correcto, se consigue una mejora. Para ello, **Clang ofrece 6 secuencias de passes** con distinto nivel de optimización, consiguiendo todos una **mejora sobre cualquier programa** recibido. Estas secuencias fueron nombradas, de menor a mayor nivel de optimización, como: **O0, O1, O2, O3, OS y OZ**. Cada una de ellas tiene 250 transformaciones aproximadamente, a excepción de O0. En ocasiones, se usará el nombre de “OX” para referirse a ellas.

2.1.4. Backend: Generación de código

Tras aplicar las transformaciones pertinentes, solo nos queda generar un código máquina que pueda ser ejecutado por la arquitectura pertinente. De este proceso se encarga **llc**, otro de los compiladores de LLVM. Consta de tres frases [8]:

- Selección de instrucciones: Se asocian las instrucciones del lenguaje IR a las del juego de instrucciones de la arquitectura destinataria. Durante este paso se hace uso de unos registros que no corresponden con los finales, llamados virtuales. Son tan solo nombres simbólicos que el compilador usa para indicar que un valor va a ser almacenado en un registro.
- Asignación de registros: Al igual que con las instrucciones, se hace una asociación entre los registros virtuales usados y los registros reales disponibles en función de la arquitectura.
- Programación de instrucciones: Se reordenan instrucciones para obedecer a las limitaciones de la arquitectura.

2.1.5. Uso de llc y opt frente a Clang

En la siguiente figura 5 se observan las relaciones entre varias librerías de LLVM y las herramientas opt, llc y Clang. En la parte superior, tenemos las dos librerías, libLLVMCodeGen y libLLVMipa, que son utilizadas en distintos subproyectos de LLVM para implementar funcionalidades. En el caso de llc, hace uso de la primera para generar código. En el de opt, usa la segunda para, además de realizar las optimizaciones propias de IR, realizar otras optimizaciones, tales como las interprocedurales, las cuales tienen como objetivo reducir el tiempo de ejecución analizando el código completo del programa.

En la parte inferior podemos ver como Clang hace uso de ambas librerías, el cual consigue superar en funcionalidad a opt y llc, además de presentar una interfaz más simple que estos dos anteriores. Por lo tanto, en la práctica, Clang puede llevar a cabo enteramente el proceso de compilación: preprocesado, análisis, generación de código IR, transformación y generación de código ensamblador.

Por esto, a Clang se le considera en la propia documentación como el "compilador nativo".

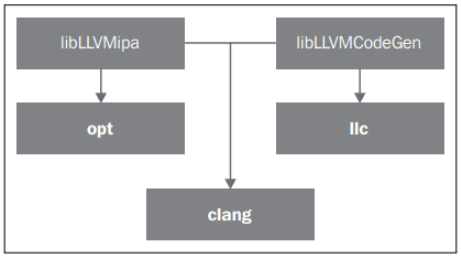


Figura 5: Relaciones entre librerías y herramientas de LLVM. [19]

2.1.6. LLVM IR, características

Previamente, se ha mencionado a LLVM IR, el lenguaje de representación intermedia propio de LLVM. Este posee una serie de características [14] las cuales se describen a continuación:

- Diseño sencillo, eficiente y fácil de entender.
- Diseño independiente de la arquitectura final y del lenguaje del que se ha convertido.
- Ofrece una gran variedad de transformaciones de distintos tipos (como se vio previamente).
- Utiliza un juego de instrucciones virtual (que no corresponde a ningún juego existente), con una semántica independiente de la arquitectura.
- Hay una propiedad llamada **SSA**, Static Single Assignment, presente en este lenguaje. Esta dice que a cada variable solo se le debe asignar un valor una única vez, además de que una variable debe ser definida antes de su uso. Esta propiedad solo se aplica, en el caso de LLVM, para todas las operaciones excepto las que asignan en memoria. Existen variantes de esta propiedad que ayudan a solucionar algunas de las complejidades que esta conlleva. [3] [8]
- Tiene definición de tipos, y los tipos son comprobados rigurosamente para asegurar la consistencia.

2.2. Benchmark

A la hora de comparar optimizaciones y transformaciones entre ellas, ya sean obtenidas o las ya establecidas, necesitamos asegurarnos de que usamos el programa adecuado, no cualquier programa es válido. Puede que el que usemos tenga pequeño margen de mejora de tiempo de ejecución debido a su simpleza, o que se vea afectado más por un tipo de optimizaciones que otro, lo cual repercutiría hacia qué optimizaciones escoger dentro del lenguaje, y una infinidad de razones más.

Debido a estos múltiples problemas, necesitamos asegurarnos de que trabajamos bajo un "estándar" establecido sobre el que podemos asegurar que los resultados obtenidos no se vean alterados. Para ello, existen los llamados Benchmark.

Un benchmark es un programa o conjunto de programas preparados específicamente para realizar mediciones de rendimiento en computadores y/o hacer comparativas entre ellos a partir de los resultados obtenidos. Como es evidente, estas mediciones pueden estar enfocadas a un componente en concreto, como medir el rendimiento de CPU o GPU, o bien una función en concreto de alguno de estos, como por ejemplo, la medición de cuántas operaciones de coma flotante puede realizar una CPU.

Hay siete características que todo benchmark ha de cumplir [10]. Estas son:

1. Relevancia: Estos han de medir características resaltables del equipo.
2. Representatividad: Deben de realizar mediciones ampliamente aceptadas por la industria y el mundo académico.
3. Equidad: Todos los sistemas han de ser comparados justamente.
4. Repetibilidad: Sus resultados han de ser verificables.
5. Rentabilidad: Deben de suponer una baja carga que procesar para el equipo.
6. Escalabilidad: Podrán ser ejecutados desde en un equipo hasta en múltiples servidores.
7. Transparencia: Las métricas obtenidas han de ser legibles y fácilmente comprensibles.

2.3. Procesamiento de lenguaje natural

Durante la historia de la humanidad, la información ha sido conservada de múltiples formas: al inicio con pinturas, más tarde mediante la escritura en papiros los cuales fueron reemplazados por libros, hasta llegar a como conocemos hoy en día. Cada una de estas formas ha presentado una serie de problemas, entre el que destaca por encima de todos: la conservación de los documentos. Existen motivos varios, entre ellos, destaca la limitación de vida útil de los materiales usados.

Con la expansión de la tecnología, y debido al exponencial crecimiento del volumen de información, nació la tarea de almacenar la información. Debido a esta tarea, aún en proceso, aparecieron múltiples campos de estudio e investigación en relación a la información ya procesada:

- Organizarla, crear índices y facilitar su acceso mediante búsquedas, lo cual dio lugar al campo de la **recuperación de la información**.

  Gracias a este, conocemos hoy en día, de manera global, los buscadores web que, sobre una consulta dada por el usuario, se recupera un conjunto de documentos ordenados en función de la relevancia que estos presentan con respecto de la consulta.

  Para ello, un método usual es crear un índice llamado invertido, donde se representa al conjunto de las palabras mencionadas en los documentos. Para cada palabra, se ha de saber en qué documentos aparece, y cuánta importancia tiene esa palabra en cada documento en la que aparece, con respecto del conjunto de documentos. Esta importancia es llamada TF-IDF. Se muestran los documentos donde aparezcan todas las palabras de nuestra consulta ordenados por la suma total de TF-IDF.

- Estudiar la interacción entre el lenguaje natural que usamos (presente en los documentos procesados) y las computadoras para, mediante el uso de inteligencia artificial, lograr múltiples objetivos antes impensables. Es lo que conocemos como **NLP**, **procesamiento de lenguaje natural**.

  Algunos de sus fines son: comprensión del lenguaje natural, generación de lenguaje natural, reconocimiento y síntesis del habla, síntesis de documentos, clasificación de documentos y detección de sentimientos.

  En este caso, la finalidad que atañe es la de **generación**. Para ello, previamente **las palabras de nuestro lenguaje han de ser codificadas**, de forma que podamos realizar operaciones matemáticas con ellas.

2.3.1. Algoritmos de procesamiento de lenguaje natural: redes neuronales

Al inicio, muchos algoritmos de procesamiento se basaban en el trabajo manual de lingüistas, los cuales hacían reglas basadas en la gramática del lenguaje y heurísticas que ayudaban al proceso de lematización, la extracción de la raíz de la palabra.

Entre finales de los años 80 y principio de los 90, debido a la "revolución estadística de la computación", fueron rápidamente sustituidos por algoritmos de **aprendizaje automático, o machine learning**. En definitiva, estos analizan un corpus, conjunto de documentos de un mismo lenguaje, y extraen las reglas por inferencia estadística.

Este presenta múltiples ventajas frente al sistema de reglas inferidas, entre las que destaca el aprendizaje automático a partir de unos datos de entrada, lo cual supone una increíble reducción de tiempo de trabajo. Además, al no tomar decisiones mediante el uso de reglas, que siempre se aplican de igual modo como es evidente, presenta una mayor flexibilidad. De este modo, los resultados que producen son más confiables, hasta el punto de que en aquellos años se hizo célebre una frase de Frederick Jelinek, investigador de NLP entre otros campos, que dijo: "Cada vez que despido a un lingüista, el rendimiento de mi reconocedor del habla aumenta".

A día de hoy, existen múltiples enfoques desde los que realizar NLP. Uno de los más usados es el de **redes neuronales**. Este es un modelo computacional de aprendizaje automático basado ligeramente en el comportamiento de las neuronas a la hora de transmitir impulsos y comunicarse entre ellas. [4]

Este modelo está formado por unos elementos llamados neuronas. Estas neuronas se organizan en capas, las cuales mantienen una relación de orden entre ellas. En cada capa pueden haber tantas neuronas como se quieran. A la primera se le llama "de entrada", a la última "de salida", y a las intermedias como "ocultas". Con respecto a las neuronas, es importante saber que cada una de ellas está conectada con todas las de la capa siguiente. A continuación, se ilustra un ejemplo (figura 6):

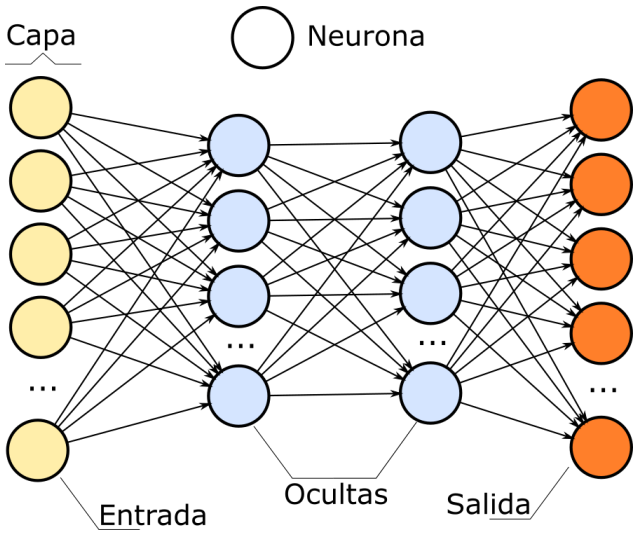


Figura 6: Ejemplo de una red neuronal.

Esta red neuronal tiene múltiples usos, pero de manera generalizada, para una determinada información de entrada dada, devuelve una salida correspondiente. Para que la red sepa qué salida ha de devolver en función de la entrada dada, ha de pasar por una etapa llamada "entrenamiento", en la que esta "aprende" a partir de unos datos que nosotros le proveemos, a los cuales se les llama patrones, que no son más que la descripción numérica de un objeto cualquiera, formado a su vez por características, valores que definen las propiedades de un objeto. Este algoritmo de entrenamiento tiene por nombre **backpropagation**.

Para su correcto funcionamiento, hemos de preparar previamente los datos. Estos deben de estar divididos en dos partes principales, las cuales llamaremos X e Y. X estará formado por todos los patrones de entrada que reciba nuestra red (patrón de entrada), mientras que Y será la salida que debe tener la red para cada patrón de X.

El objetivo del algoritmo es, dado los patrones de X, minimizar el error producido entre los patrones de Y, la salida que se desea que tenga la red, y la salida que la red produce.

Su funcionamiento se divide en dos etapas:

- Fase de propagación: Donde se aplica un patrón de X a la capa de entrada, los cuales se propagan hasta la última capa. Para la propagación, cada neurona produce una salida para cada neurona a la que está conectada.

La salida de una neurona i con respecto de otra j, a la que está conectada, es:

$$o_{i,j} = f_i(x_i * W_{i,j}) \tag{1}$$

Donde:

- $o_{i,j}$ es la salida de la neurona i con respecto de la j. La neurona j toma esta salida como parte de su entrada.
- f_i es una función llamada de activación, definida por el usuario. Cada neurona posee una que puede ser o no diferente de la del resto. Si la red neuronal tiene varias capas, conocida usualmente como multicapa, esta debe de ser una función no lineal. De lo contrario, la red puede ser reducida a otra de una única capa, obteniendo el mismo resultado.

$$W3 * (W2 * (W1 * X))) = (W3 * W2 * W1) * X \tag{2}$$

- x_i es la entrada de la neurona i.
- $W_{i,j}$ es un valor asociado a la conexión entre la neurona i y la j, llamada "peso". Conceptualmente, si agrupásemos todos los pesos para cada neurona j existente con la que está conectada i, tendríamos el peso W_i de la neurona i.

Una vez se obtiene la salida de la capa de salida, se compara esta con la salida deseada, el patrón correspondiente en Y.

- Fase de adaptación: Se calcula el error cometido, el cual se distribuye en sentido inverso por toda la red, e individualmente cada neurona modifica su peso en función del error. De este modo, podemos decir que la red "aprende" con cada patrón al variar el peso, siendo estos pesos las variables que reflejan el entrenamiento, luego son de suma importancia. Las modificaciones pertinentes se calculan:

$$w_{i,j} = w_{i,j} - \mu \cdot \frac{\partial E}{\partial w_{i,j}} \tag{3}$$

Donde:

- $w_{i,j}$ es el peso entre las neuronas i y j.
- μ es un valor llamado paso. Este se utiliza debido a que el aprendizaje del backpropagation está basado en el gradiente descendiente, un algoritmo de estimación de los parámetros de una función donde se encuentra esta variable, la cual va decreciendo conforme aprende la red. Su valor refleja cuánto se desplaza la solución actual obtenida con respecto del espacio de error, y evidentemente, también muestra cómo cambian los pesos de la red. Es un valor con suma importancia a definir para un correcto aprendizaje. Si es muy grande, la solución presentará un efecto oscilatorio sin llegar a conseguir una solución óptima. Si es demasiado pequeño, la solución no cambiará. Generalmente es conocido como **learning rate** (tasa de aprendizaje).
- $\frac{\partial E}{\partial w_{i,j}}$ es la derivada del error total obtenido con respecto del peso antes mencionado. Esta derivada se desarrolla de diferente forma para la capa de salida en comparación con las ocultas. [4]

Además, por último, es importante destacar que el entrenamiento explicado es de un tipo nombrado de **"aprendizaje supervisado"**. Es así ya que, por cada salida generada, existe una salida deseada con la que poder comparar y corregir en consecuencia. Del mismo modo, existe aprendizaje no supervisado en otros modelos computacionales.

El modelo, como se ha observado, es altamente funcional. Sin embargo, con respecto de nuestro problema, presenta una serie de problemas. A continuación, en el siguiente apartado, se explican.

2.3.2. Redes neuronales recurrentes

El problema que presenta las redes neuronales tradicionales ante los problemas de NLP es la falta de persistencia. Tanto en el lenguaje, como en general los seres humanos al procesar la información, no procesamos cada elemento de manera individual, sino que conservamos la información en pos de procesar otra posterior. En el procesado de lenguaje natural ocurre algo similar. Dada, por ejemplo, la frase: "Ella se ha mudado a París. Ahora vive en", si

tenemos persistencia, podemos deducir que una posible palabra final pudiese ser "Francia". Si careciésemos de esta, la palabra final sería la siguiente tras "en", sin tomar en consideración el resto de palabras anteriores.

Para solucionar este problema, aparecieron las llamadas **redes neuronales recurrentes**, o RNN. En estas, para cada neurona, se toma como parte de la entrada la salida anterior producida, en otras palabras, la salida perteneciente al instante o timestep anterior (t-1 siendo t el instante actual). Dicho lo cual, el cálculo de la salida entre la capa de entrada y la oculta de cada neurona sería: [31]

$$o_{i,j} = f_i(x_i * W_{i,j} + U_{i,j} * o_{t-1} + b) \tag{4}$$

Siendo $U_{i,j}$ un nuevo peso adicional y o_{t-1} la salida en el instante anterior. Por otro lado, b representa el **bias**, variable que mide el grado de ajuste con respecto del conjunto de datos usados para entrenar nuestra red. Este será un concepto que aparecerá más adelante en varias ocasiones.

Se puede observar, pues, que se produce un efecto de retroalimentación. Podemos ver una representación de una red neuronal recurrente de una sola capa en la figura 7.

Es importante recordar que, al igual que en las redes neuronales clásicas, cada neurona de capa está conectada con todas las neuronas de la capa siguiente, aunque no siempre debe de ser así. Con este cambio, además, las neuronas de una misma capa están conectadas entre sí, una a una.

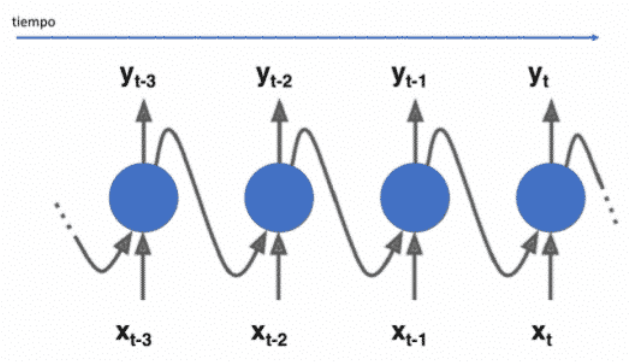


Figura 7: Ejemplo de una red neuronal de una neurona, desglosada en distintos instantes de tiempo. [31]

En los últimos años, se ha usado exitosamente en una variedad de problemas. Dado que la salida de una neurona depende de otra de un instante anterior, se suele decir que las neuronas tienen memoria, en cierta forma. Al hecho de que una neurona preserva información, esa memoria, se le llama "memory cell". Es por esta razón que, como se observa en el ejemplo dado al inicio de este subapartado, este modelo es muy adecuado para problemas de NLP y categorización, entre otros.

A raíz de la división temporal en instantes de tiempo, aparecen distintas variantes en la arquitectura, en función del input recibido y el output deseado. Cada uno de ellos, de manera general, suele usarse en un tipo de problemas. En la figura 8 se muestran cada uno de ellos de forma gráfica. A continuación, se explica cada uno de ellos y se detalla su uso:

- One to one (one input, one output): Una entrada, ya sea una palabra en el caso de NLP o una imagen otro tipo de problemas, es recibido y la red produce una sola salida. Un ejemplo de su uso es la clasificación de imágenes.
- One to many (one input, many outputs): Una sola entrada y muchas salidas, donde hay tantos elementos en la capa oculta como salidas. Como podemos observar, hay más elementos en la capa oculta que entradas, debido a que se requieren más salidas. Pese a no haber entrada para cada neurona de capa oculta, como mencionamos previamente, se recibe como entrada la salida generada anteriormente. Se suelen usar para "image caption" (generar la descripción de una foto), donde se da una imagen, y se devuelve una secuencia de palabras.
- Many to one (many input, one output): Muchas entradas, una salida. En esta arquitectura se evidencia que no todas las neuronas de la red tienen que producir salida. Un ejemplo de su uso, el cual se relaciona con nuestro problema, es la predicción de la última palabra de una frase. También es usado en el análisis de sentimientos.
- Many to many (many inputs, many outputs) con "desplazamiento": Se tienen más elementos de capa oculta que de entrada y salida. El número de entradas y el de salidas no tiene porqué ser igual, pese a como está

representado en la figura. Se suele usar para traducción entre idiomas, donde se da una frase en un idioma (capa de entrada), se procesa (capa oculta), y se devuelve la frase en otro idioma (capa de salida).

- Many to many (many inputs, many outputs): Generalmente usado en clasificación de vídeo, donde se etiqueta frame a frame, las entradas son los frames, las salidas las etiquetas, y hay tantas entradas como salidas.

Es importante aclarar que, tanto en la explicación como en la figura, las entradas y salidas pueden ser una variedad de elementos, como una imagen o una palabra, todos ellos codificados numéricamente. Por ejemplo, una palabra puede estar representada por un vector de 10 valores numéricos. Tanto entradas, salidas, como elementos de capa oculta están representados en la figura por un rectángulo. Estos rectángulos pueden equivaler a una neurona.

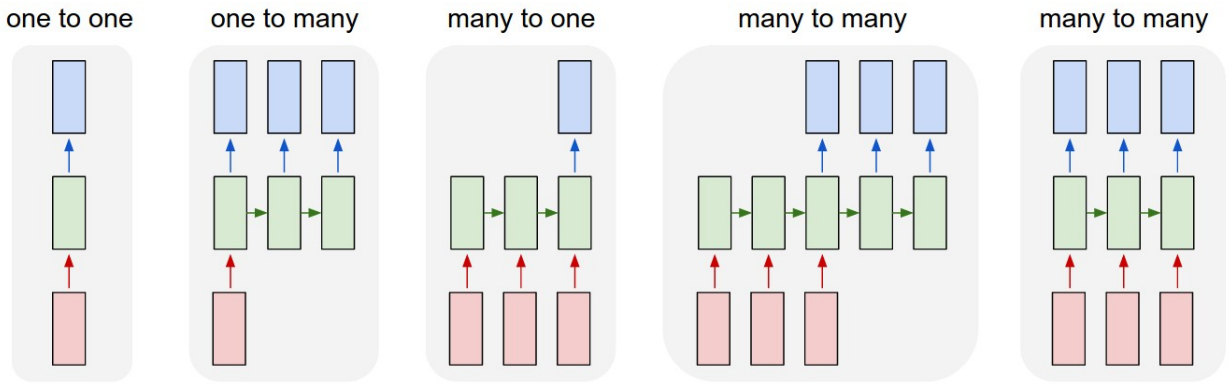


Figura 8: Arquitecturas de RNN, siendo cada rectángulo un elemento, input (rojo), output (azul), o elementos de capa oculta (verde). [13]

Como podemos ver por la figura 8, la arquitectura general de la red y el modo en el que las neuronas calculan su salida cambia, al tener en cuenta como input la salida de otra neurona, además de la conexión con la siguiente capa. A raíz de esto, el algoritmo de backpropagation no puede aplicarse del mismo modo. El que se usa con este tipo de redes es una variante de este, llamada **backpropagation throught time, o BPTT**, backpropagation a través del tiempo (donde con los tiempos se hace referencia a los timesteps antes mencionados). A continuación comentaremos el funcionamiento de sus fases:

- Fase de propagación: Como se ha comentado previamente, en la capa oculta, el input de cada neurona tiene en cuenta, además, la salida de la neurona anterior. Decimos, por esto, que las salidas se desplazan hacia la derecha de la red. Por lo demás, como se comentó con anterioridad, se procede de modo normal.

- Fase de adaptación: Al existir "desplazamientos laterales" entre las salidas (la salida de una neurona es entrada de otra), se hace evidente que el algoritmo no puede aplicarse del mismo modo, donde cada neurona calcula su error independientemente del resto. En esta fase, cada neurona de la capa oculta tiene en cuenta los errores cometidos por las neuronas previas en su salida, al ser recibida como entrada, además de su propio error. Este cambio se refleja en la derivada del error con respecto del peso, donde, sin entrar en la formulación paso a paso, queda tal que:

$$\frac{\partial E_n}{\partial W} = \sum_{k=0}^{n} \frac{\partial E_n}{\partial y'_n} \frac{\partial y'_n}{\partial s_n} \frac{\partial s_n}{\partial s_k} \frac{\partial s_k}{\partial W} \tag{5}$$

  Siendo:

  - s_n la salida entre la capa de entrada y la oculta de una neurona n.
  - $y'n$ la salida producida por la siguiente capa a esta.
  - E_n el error producido por la salida correspondiente a esta neurona.
  - W el peso entre la neurona y la de la siguiente capa.

  Como se puede observar, la derivada del cálculo requiere de la derivada de s_k, la salida de todas las anteriores neuronas. Por lo tanto, se puede decir que para el cálculo de error, y por lo tanto la adaptación, de una neurona se toma en cuenta los resultados previamente calculados (figura 9).

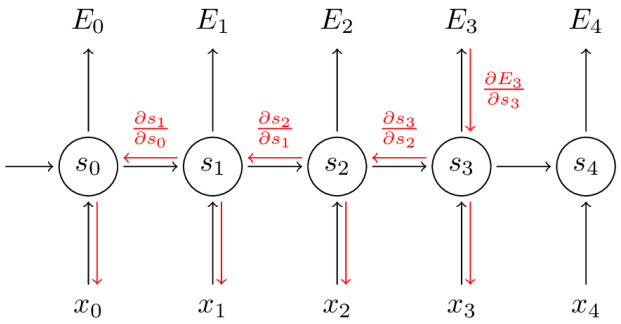


Figura 9: Cálculo del error de una neurona E_3, donde se representa la implicación de las salidas de neuronas previas. [6]

2.3.3. LSTM

Como comentábamos previamente, debido a la retroalimentación de las RNN (la salida de una neurona se toma como entrada de la del timestep siguiente) decíamos que estas redes poseían "memoria", comúnmente llamada memory cell. Esta les posibilita mantener información a lo largo del tiempo. Sin embargo, este mantenimiento tiene sus limitaciones. Existen problemas llamados **long-term dependency**, o de larga dependencia de términos, difíciles de abarcar con el uso de este tipo de red.

Como su propio nombre indica, es un tipo de problema donde debemos de mantener la información de un término dado durante un largo periodo de timesteps indeterminado. Para facilitar la comprensión de estos problemas, pongamos un ejemplo:

Yo nací y me crié en **Italia**. [...] Hablo de manera fluida el **italiano**.

En este ejemplo, nos encontramos frente a un problema de generación de la siguiente palabra dada una frase. Como se observa, la necesidad de recordar el término Italia es importante para más adelante poder predecir que la siguiente palabra a "el" sería "italiano".

Prosiguiendo con el RNN, es difícil para esta red trabajar con este tipo de problemas. Se debe a que el gradiente de la función de pérdida, representado con el uso del paso, decae exponencialmente con el tiempo, problema nombrado como **desvanecimiento del gradiente descendente**. Debido al desvanecimiento del gradiente, surge una de las múltiples variaciones de RNN, llamada **LSTM, o Long Short-Term Memory**, que lo soluciona.

En las LSTM aún se encuentra presente el problema de desvanecimiento, pero ocurre en una menor medida. A pesar de ello, sopesa la ya mencionada "larga dependencia de términos" con la adición de un nuevo término, que veremos más adelante.

Las redes LSTM modifican la funcionalidad de las neuronas de la capa oculta de las RNN estándar. En estas, se añade un nuevo campo a distribuir entre neuronas, llamado "cell state", que representa la memoria de las neuronas. Además, cada neurona contiene dentro una serie de funciones, llamadas "puertas", cada una con una función específica. En cada una de ellas hay un valor de peso y uno de bias. A continuación, se procederá a describir cada una de ellas en el orden en las que se ejecutan.

Antes de eso, es importante aclarar que, junto a la explicación de cada una de ellas, está presente una figura. Todas ellas usan la notación mostrada en la figura 10.

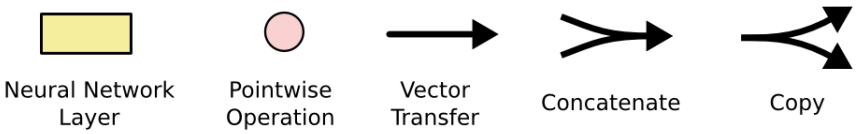
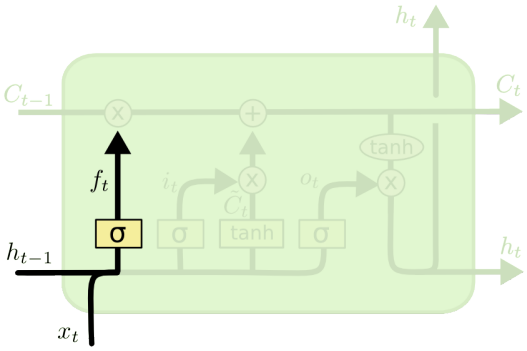


Figura 10: Notación usada en las figuras descriptivas de LSTM. [25]

Siendo, además, W_x el peso correspondiente a la puerta x, b_x el bias de la puerta x (siendo x la letra inicial de su respectivo nombre en inglés), h_{t-1} la salida de la neurona anterior (por lo que la salida de la actual es h_t), y C_{t-1} el cell state de la neurona anterior (y C_t el actual). σ corresponde a la aplicación de la función sigmoide, y tanh la función tangente hiperbólica. Una vez aclarado, se procede con la explicación de las puertas, y por lo tanto, el funcionamiento de las neuronas:

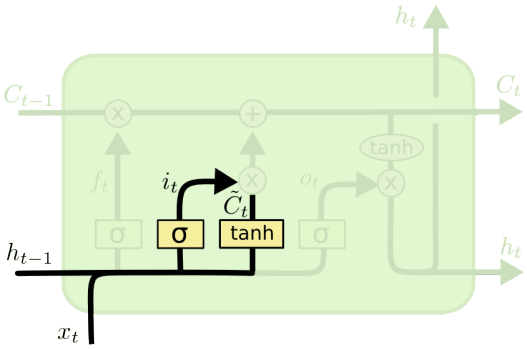
- Forget gate (puerta de olvido, figura 11): En esta, se decide qué información conservar y qué información desechar, por lo que solo se modifica el cell state. Para ello, se concatena las entradas recibidas, h_{t-1} de la neurona anterior y x_t de la capa anterior. Tras ello, se multiplica por el peso y se suma el bias, para finalmente aplicar la función sigmoide. De este modo, obtendremos un vector, nombrado como ft, con valores entre 0 y 1, siendo 0 información a olvidar, y 1 información a recordar. Tras esto, aún no se actualiza el cell state.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figura 11: Forget gate. [25]

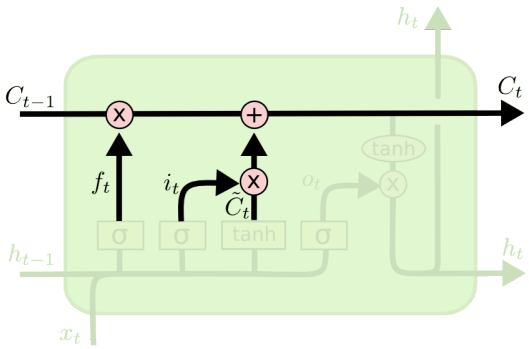
- Input gate (puerta de entrada, 12): A continuación, se decide qué información de la recibida se va a almacenar en el cell state. Para ello, haremos varios cálculos.
  - Primero, calculamos un vector i_t, de posibles valores a actualizar en el estado, en otras palabras, cuáles queremos actualizar. Para ello, usamos la función sigmoide sobre la concatenación de h_{t-1} y x_t, tras multiplicar y sumar. El vector resultante valores entre 0 y 1.
  - Más tarde, haremos un vector de posibles candidatos a añadir (sumar) al cell state, con el uso de la función tangente hiperbólica, por lo que este vector tiene valores entre -1 y 1.
  - Estos dos son multiplicados para formar una actualización del cell state. De esta forma, obtenemos un vector donde los valores de los candidatos están escalados en función de cuánto, de 0 a 1, queremos actualizarlos.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Figura 12: Input gate. [25]

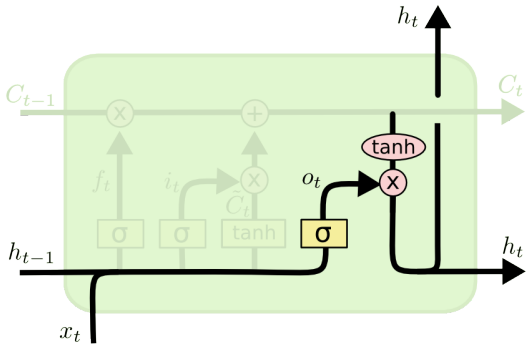
- Actualización del cell state (figura 13): Con la información recabada por los dos anteriores pasos, se realiza la actualización. Para ello, se multiplica el cell state anterior, C_{t-1}, por el vector f_t, que decidía qué datos debemos mantener u olvidar. Tras ello, aplicamos la actualización calculada anteriormente, sumando el vector de actualización y el cell state. Este cálculo se puede realizar en cada una de los pasos anteriores por separado, en vez de todo junto, con igual resultado.
- Output gate (puerta de salida, figura 14): En esta puerta generamos un output para la neurona de la capa superior (línea vertical con h_t) y para la siguiente neurona (línea inferior horizontal con h_t). Para ello, generamos el vector o_t para decidir qué valores del cell state vamos a devolver, con el uso de la función sigmoide (0 si no se devuelve, 1 si sí). Este vector se multiplica con el cell state, filtrado por una función tangente



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figura 13: Actualización del cell state. [25]

hiperbólica. Si nos fijamos, hasta antes de este paso toda la información relevante está en el cell state, incluso el output se genera a partir de él. Con este filtro, nos aseguramos que la salida h_t queda con valores entre -1 y 1.



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figura 14: Output gate. [25]

Una vez explicado el funcionamiento individual de las neuronas, ¿cómo mitiga la red el problema del desvanecimiento del gradiente? Bien pues, esta mitigación viene dada por el uso de la estructura regulada (reguladora a raíz debido al uso de funciones que deciden qué actualizar, olvidar, devolver, ...) de puertas presente en las neuronas, que condiciona la distribución del error. Del mismo modo, la división de pesos en las distintas puertas y la diferenciación con respecto a la operación de las RNN estándar que se aplica de forma sucesiva, ayudan del mismo modo. En el caso de este segundo, ayuda a estabilizar la memoria, como es evidente.

2.3.4. LSTM bidireccional

Existen múltiples variantes de las LSTM originales. Entre ellas, se encuentra la LSTM bidireccional, o BDLSTM, o BiLSTM. Esta es, en realidad, una variante de las RNN conocida como BRNN, pero también existe su variante en LSTM.

Su principal característica es el uso de dos capas ocultas, en vez de una. En la primera de ellas, como se ha explicado previamente, las salidas de cada neurona son parte de la entrada de la siguiente neurona, desde la primera hasta la última. Sin embargo, la otra capa tiene un funcionamiento diferente. En concreto, tiene una dirección diferente. Se comienza a procesar con la última neurona, la cual proporciona su salida a la penúltima, la penúltima a la antepenúltima, y así sucesivamente hasta llegar a la primera.

De esta forma, podemos decir que en la LSTM bidireccional tenemos dos capas ocultas con dos direcciones distintas: en una se procesa de inicio a final, hacia delante, y otra de final a inicio, hacia atrás.

Cada output se genera a partir de las salidas de ambas capas, teniendo en cuenta tanto el "pasado" (las salidas empezando por la primera neurona) como el "futuro" (las salidas empezando por la última neurona). Se usa una función para ello, que combina las salidas de ambas neuronas. Puede ser una función de concatenación, de suma, de multiplicación, o una función que calcula la media entre los valores de ambas salidas.

En la figura 15 se muestra una representación gráfica de la red.

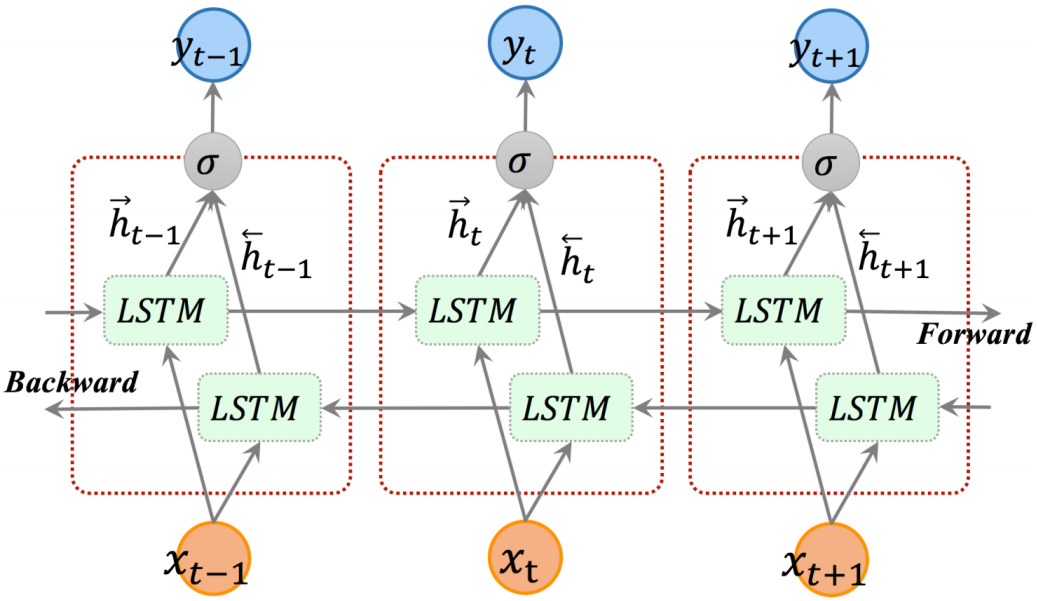


Figura 15: Funcionamiento de una red LSTM bidireccional, donde el bloque "LSTM" representa una neurona. [9]

Siguiendo la notación de la figura, el cálculo de la salida sería:

$$y_t = \sigma(h_t^{\rightarrow}, h_t^{\leftarrow}) \tag{6}$$

Este cambio de dirección en una de las capas hace que el entrenamiento requiera pasos adicionales. Para comenzar, el primero es cómo generar la salida de las capas ocultas, que se acaba de detallar. El otro sería, como se explicó en BPTT, a la hora de propagar el error. Este iría en la otra dirección, debido a las derivadas de los errores.

2.4. Algoritmos de codificación: Word2Vec y su importancia

Una vez explicadas las herramientas a usar en nuestro problema NLP, únicamente resta un único gran problema. Como hemos visto anteriormente, las redes operan matemáticamente con números, vectores, o matrices (estos dos desde el punto de vista informático). De este modo, no podemos utilizar las palabras como tal. Para ello, existe un proceso llamado codificación en el que se obtiene una representación numérica para cada palabra de nuestro vocabulario, siendo el vocabulario el conjunto total de nuestras palabras.

Una de las representaciones más sencillas desde el punto de vista de la implementación, es la llamado "one-hot encoding". En esta, se le otorga un valor único a cada palabra del vocabulario. Para codificar, o en otras palabras, obtener la representación de una palabra, se crea un vector de tantas posiciones como palabras tenga el vocabulario, y se coloca un 1 en la posición correspondiente a su valor único. De esta forma, una frase está formada por un secuencia de vectores. Podemos ver un ejemplo en la figura 16.

De un modo similar, existe la codificación "index-based", o basada en la codificación. En esta, en vez de tener una serie de vectores binarios, tenemos un único vector por frase, con tantas palabras como tenga la frase, donde cada posición refleja qué palabra hay, mediante el identificador único. A continuación, se muestra una codificación de ejemplo en la figura 17.

En lugar de estas dos, usaremos otra codificación, llamado word vector. En este, se crea un espacio n-dimensional, donde cada palabra ocupa un lugar concreto. Este lugar concreto de ese espacio de n dimensiones es el word vector, donde cada valor es el lugar que ocupa en cada dimensión de las n totales, por lo que el vector tiene n posiciones. Esto nos ofrece una serie de ventajas frente a los otros modelos:

- En la codificación one-hot, cuantos más palabras tenemos en el vocabulario, más crece el vector correspondiente a cada palabra. Esto, sumado al hecho de que la mayoría de posiciones son 0, lo hace altamente ineficiente. En comparación, los word vector son de tanta dimensión como se desee, mejorando notablemente la eficiencia al poder hacerlos de menor tamaño, y sobretodo, un tamaño fijo.

	La	casa	blanca	es	la	mejor
0	**1**	0	0	0	1	0
1	**0**	0	1	0	0	0
2	**0**	1	0	0	0	0
3	**0**	0	0	1	0	0
4	**0**	0	0	0	0	1

Donde cada palabra corresponde al índice:

La	casa	blanca	es	mejor
0	2	1	3	1

Figura 16: One hot enconding.

La	casa	blanca	es	la	mejor	de	todo	el	barrio
0	2	1	3	0	**4**	5	7	6	8

Figura 17: Index-based encoding.

- Los word vector tienen la capacidad de **generalizar**. Los conceptos con alta relación semántica, en otras palabras, un alto significado en común, aparecerán juntas.

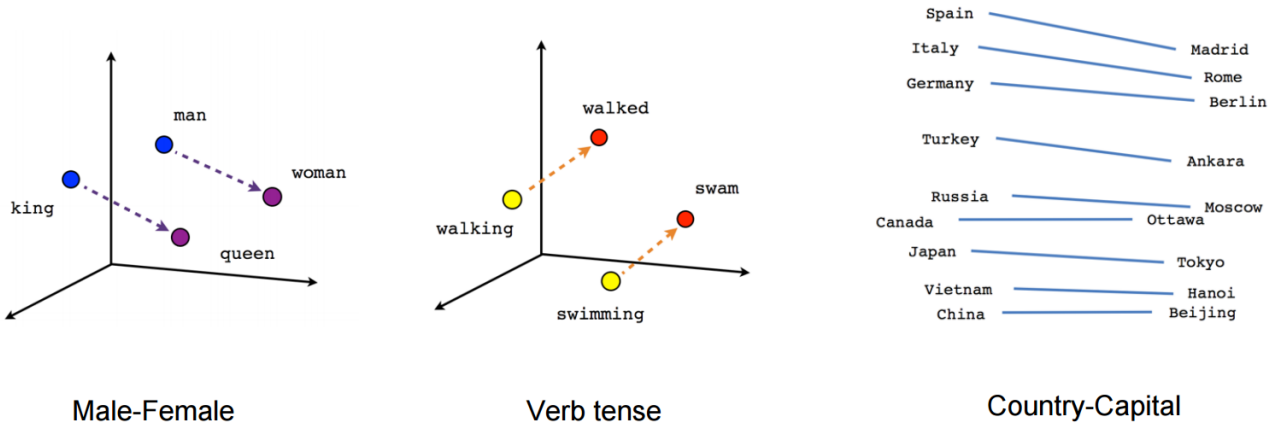


Figura 18: Relación entre word vectors de semántica común.[27]

En la figura 18, podemos ver un ejemplo. En esta, verbos en distintos tiempos verbales aparecen juntas.

Del mismo modo, términos con igual relación entre ellos distan la misma distancia entre ellos, como podemos ver en el ejemplo de las capitales o en el que solo cambia el género de las palabras.

Esto nos permite una propiedad interesante, y es la de realizar analogías entre palabras, usando las operaciones entre ellas. Para ello, pondremos el ejemplo de la figura anterior. "King" es a "man", como "queen" es a "woman". De este modo, si al word vector de "King" le restamos el de "man" y le añadimos el de "woman", el resultado será el de "queen" (figura 19).

Del mismo modo, ya que su relación se basa en la semántica (relaciones), podemos observar que son de alta utilidad para tareas de traducción. Al tener el mismo significado, independientemente del idioma, posee el mismo lugar en el espacio n-dimensional, y posee las mismas relaciones (figura 20).

Como se observa pues, las propiedades de los word vector presentan una multitud de propiedades que son de alta utilidad de cara a un posterior uso junto a algún algoritmo de NLP, como el previamente explicado LSTM. La

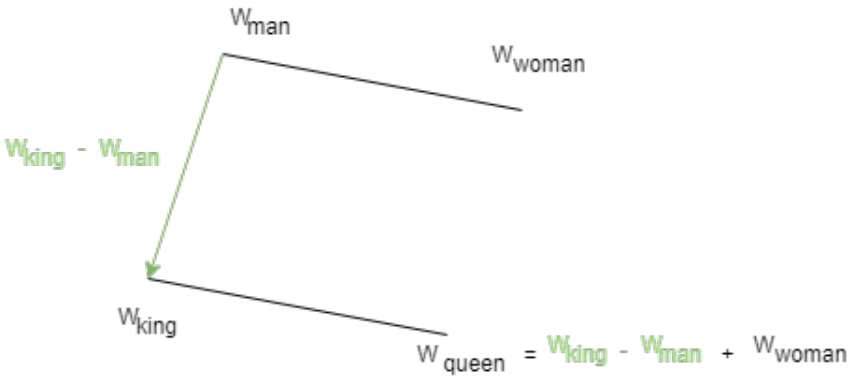


Figura 19: Analogía entre word vectors.

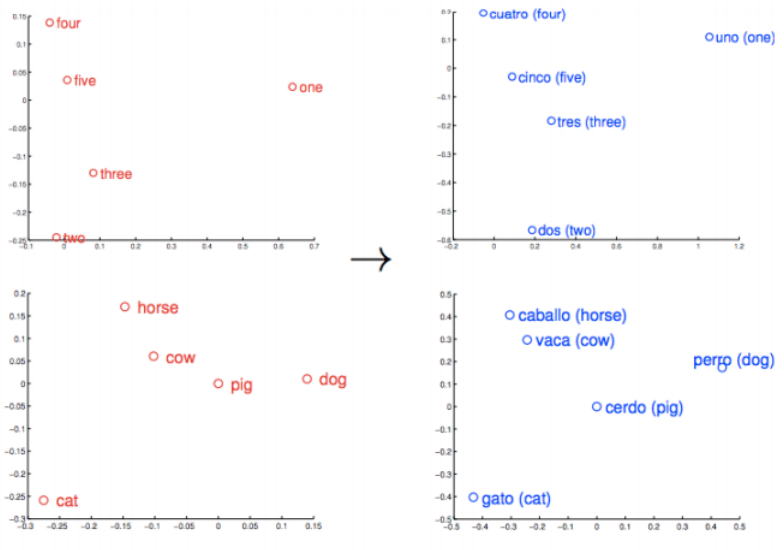


Figura 20: Word vector aplicado a traducción.[24]

representación de cada palabra da mucha más información de la misma que el resto de representaciones explicadas, y recordando la definición de que un patrón está formado por características, propiedades de un objeto, se hace evidente que es de utilidad.

Sin embargo, los word vector presentan un gran problema, que es la **polisemia**. Dado que solo existe un word vector por término, y existen multitud de términos con distinto significado y por lo tanto distintas relaciones, podemos obtener una representación errónea.

Más tarde veremos cómo la polisemia afecta a nuestro problema a tratar.

Ahora bien, ¿cómo se generan los word vectors? Para ello existe el llamado **word2vec**, una colección de modelos computacionales que se encarga de ello mismo. Todos sus modelos se tratan de redes neuronales estándar de tres capas: entrada, oculta, y de salida.

Al igual que con las redes neuronales tradicionales, tenemos que preparar los datos para su posterior entrenamiento. Para comenzar, como es evidente, necesitamos un corpus, en otras palabras, un conjunto de documentos. A partir de este, ¿cómo obtenemos los patrones de entrada y las salidas deseadas? De ello se encarga el mismo modelo, el cual los genera conforme va entrenando. Por este motivo, este entrenamiento se encuentra en una categoría entre aprendizaje supervisado y no supervisado, llamada **aprendizaje autosupervisado**.

Uno de los requisitos que requiere este aprendizaje autosupervisado es que definamos un valor "C", siendo C un número entero. La red funciona de la siguiente manera: itera por cada palabra del documento, siendo esta palabra denominada la "central", y escoge las C palabras a su alrededor (tanto izquierda como derecha siempre que sea posible) para entrenar. Estas palabras de a su alrededor es lo que se denomina su "contexto", por lo que C es el valor que delimita el contexto de cada palabra. Un ejemplo de ello es el siguiente:

C=2, siendo t el valor de la palabra central actual, la cual es amarilla, y verde su contexto:

Mi casa es la mejor de todo el barrio, t=0
Mi casa es la mejor de todo el barrio, t=1
Mi casa es la mejor de todo el barrio, t=2
Mi casa es la mejor de todo el barrio, t=3
...
Mi casa es la mejor de todo el barrio, t=9

Entonces, ¿la palabra central se toma como entrada, o la entrada es el contexto? Esto depende de la arquitectura de Word2Vec que escojamos. Existen dos arquitecturas a escoger, **CBOW, Continuous Bag-of-Word, y Skip-Gram**. Previamente a explicar su funcionamiento, hay que aclarar que ambas, como en toda red neuronal, no pueden usar las palabras directamente como entrada. Por ello, se utiliza un conveniente codificado de one-hot vector.

- CBOW: Tiene como entrada las palabras de contexto con respecto de la palabra general, es decir, C palabras en total. Cada palabra, recordemos, está codificada como one-hot vector de tamaño V, siendo V el tamaño del vocabulario. Por lo tanto, en resumen, tenemos como entrada C vectores de tamaño V, en total, CxV valores, y por lo tanto, CxV neuronas.

  Para facilitar su comprensión, podemos suponer que a cada vector de entrada le corresponde una matriz de peso de tamaño VxN, donde N es las dimensiones de nuestro espacio, y por lo tanto, el tamaño del vector que representa a una palabra.

  Para generar la salida de la capa de entrada, se multiplican los vectores por sus correspondientes matrices de peso. Tras esto, se calcula la media entre todos los vectores resultantes para obtener el vector "h", de dimensión 1xN. Por lo que tenemos N neuronas en la capa oculta.

  Finalmente, para calcular la salida de la capa oculta, se multiplica el vector "h" por la matriz de peso correspondiente, de dimensión NxV, para más tarde aplicar una función de activación softmax. Así, se obtiene el vector "y", de dimensión 1xV, el cual representa en cada índice la probabilidad de escoger cada palabra del vocabulario, de ahí la razón de su dimensión. Teniendo así V neuronas en la capa de salida.

  Sabiendo qué palabra deberíamos de escoger y cuál es la más probable según "y", se calcula el error y se corrigen los pesos. El error no es más que la diferencia entre entre la salida generada, "y", y la deseada, el vector de la palabra central.

  El objetivo es intentar predecir la palabra central a partir del contexto.

  La codificación final queda representada en la matriz de peso entre la capa oculta y la de salida (de dimensión NxV), donde cada columna (de dimensión N) es la codificación de una palabra, de ahí que hayan V columnas.

  Todo ello queda representado en la figura 21.

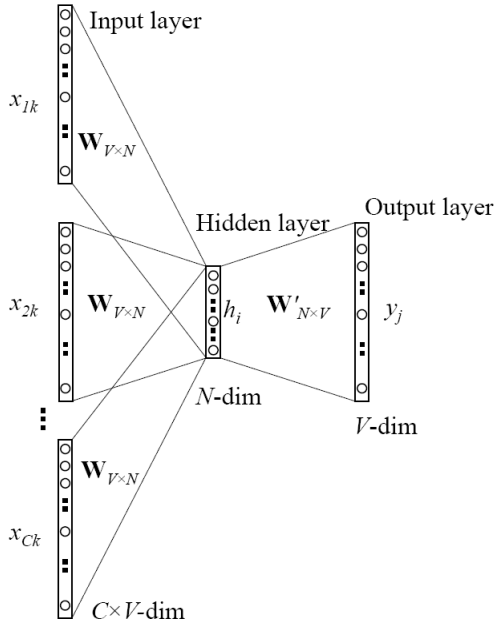


Figura 21: Arquitectura CBOW.[28]

- Skip-Gram: Tiene como entrada la palabra central, es decir, un one-hot vector de dimensión 1xV, V neuronas. Como salida de la capa de entrada, se realiza una multiplicación de este vector y la matriz de peso correspondiente. En esta matriz de peso, al igual que en CBOW, se encuentra la codificación de las palabras, esta vez en cada fila. Por ello, su dimensión es VxN. Como resultado de la multiplicación, la salida de la capa de entrada es el vector correspondiente a la codificación de la palabra central, vector que nombraremos como "h", de dimensión 1xN, teniendo N neuronas en la capa oculta.

$$[0010] \times \begin{bmatrix} 123 \\ 234 \\ 463 \\ 998 \end{bmatrix} = [463] \tag{7}$$

  Este vector h, más tarde, se ve multiplicado por tantas matrices de peso VxN como palabras haya en el contexto, es decir, C matrices de VxN. Como resultado, tenemos C vectores a los que se le aplica softmax, obteniendo C vectores de tamaño V (CxV neuronas).

  De nuevo, como antes, V es un vector que representa en cada índice cuál es la probabilidad de escoger cada palabra del vocabulario. El objetivo en esta arquitectura es predecir las palabras de contexto a partir de la palabra central.

  De esta manera, sabiendo las palabras de contexto que deben aparecer, y las C palabras predichas según los vectores, se calcula el error y se adaptan los pesos. El error no es más que la diferencia entre la salida producida, los C vectores, y la deseada, los vectores de las palabras de contexto.

  Una representación gráfica de ello puede apreciarse en la figura 22.

Ambos modelos fueron propuestos por Tomas Mikolov, científico de la computación especializado en aprendizaje automático, en un documento en el que plantea el uso de Word2Vec, sus propiedades y resultados. [21]

Asimismo, realiza un análisis de ellos, señalando que:

- Skip-Gram funciona bien para un conjunto pequeño de datos de entrenamiento, es más lento, pero consigue una mejor representación de las palabras infrecuentes.
- CBOW tiene un menor tiempo de ejecución a la hora de entrenar y consigue una mejor representación de las palabras frecuentes.

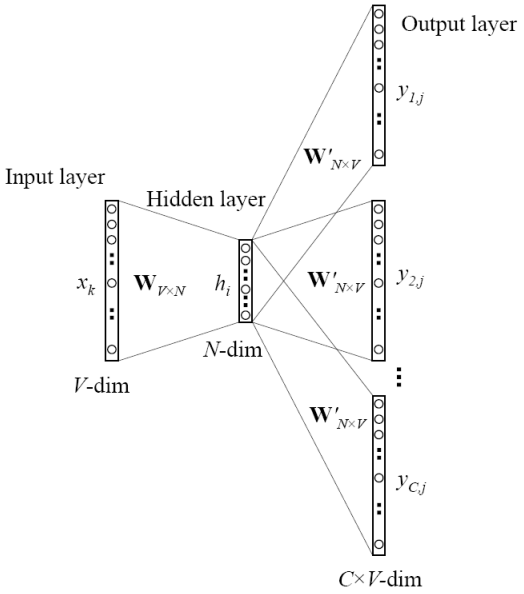


Figura 22: Arquitectura Skip-Gram.[28]

Pero estas dos arquitecturas comentadas, las cuales se han explicado tal y como su implementación original funciona, tienen posibles opciones de optimización a aplicar. Mikolov, en su documento, destaca tres métodos de optimización:

- Subsampling of frequent words (submuestreo de palabras frecuentes): Existen palabras que pueden llegar a aparecer, en un corpus de gran tamaño, cientos de millones de veces. Estas palabras proveen, indivualmente, de menor información. Tras procesarse la misma palabra múltiples veces, el cambio que se aprecia en su representación es insignificante, ya que esta se acerca a la que debe tener a partir de nuestro corpus. Como se hace evidente, si en vez de procesar una palabra como la mencionada, procesásemos otras raras, el aprendizaje sería mucho más ligero.

  De esta forma, para cada palabra que se vaya a procesar, existe una posibilidad de que esta se descarte y se procese otra. Esta probabilidad viene dada por la frecuencia de esta palabra en el corpus. Junto a la frecuencia, se usa una variable "t", conocida como límite, para ajustar la probabilidad. Suele tener un valor de 10^{-5}.

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}} \tag{8}$$

- Hierarchical Softmax (softmax jerárquico): Fue propuesto por Morin y Bengio[23], además de Mnih y Hinton [22] en 2005 y 2009 respectivamente. Este es una manera eficiente de ejecutar la función softmax, usada en ambas arquitecturas. Para aumentar su eficiencia, se usa un árbol binario donde hay V hojas, siendo V el número de palabras en el vocabulario. Hay V-1 nodos internos, y para cada hoja existe un único camino hasta la raíz del árbol. Este camino es el usado para calcular la probabilidad de cada palabra. [28]

- Negative sampling (muestreo negativo): Propuesto por Noah A. Smith y Jason Eisner [30] como "Contrastive Estimation", y más tarde reformulado por Michael Gutmann y Aapo Hyvärinen [12] como "Noise-contrastive estimation".

  Esta optimización trata de reemplazar el uso de la función softmax, debido a que esta tiene un coste computacionalmente alto cuando tenemos un gran número de neuronas.

  Para ello, se modifica el funcionamiento de la red. De este modo, en vez de enfrentar un problema de clasificación (como, por ejemplo, el de predecir la siguiente palabra dada una frase), tenemos uno de clasificación binaria. Para ello, se le otorga a la red la palabra central, una de su contexto, a la que llamaremos "positive sample" (por seguir la terminología de Mikolov), y otras k palabras sacadas de una distribución de ruido P que están fuera del contexto de la palabra central. Estas k palabras son llamadas "negative samples".

Tras ello, la red trata de predecir cuáles son las palabras positivas, y cuáles las negativas. De este modo, conseguimos de igual manera la codificación de las palabras.

Originalmente, los negative samples eran extraidos de manera aleatoria del vocabulario. Mikolov, sin embargo, modificó esto (como hemos visto al sacarlas de una distribución P), y le dio el nombre de "Negative sampling".

Finalmente, hemos de destacar que, **para cada dato de entrada dado a la red, se calcula el error y se adapta**. Hasta ahora, en la explicación dada siempre se ha concebido que es así, **pero no tiene porqué serlo**. Estas son las llamadas **variaciones del algoritmo de gradiente descendiente**. En el caso ya comentado de que se produce una adaptación por cada dato dado, se conoce como descenso del gradiente estocástico. Las otras variaciones toman un número de datos determinado, conocido como "batch" antes de adaptar la red. Es el llamado descenso en lotes. El valor de batch puede ser desde un número mayor que uno hasta el total de datos disponible en la red.

2.5. Algoritmo genético

El algoritmo genético es uno de los más usados en el campo de la inteligencia artificial, además de uno de los que tiene mayor cantidad de aplicaciones. Está basado en la evolución biológica. En este, tenemos una población ideal de individuos, los cuales son sometidos a evolucionar, mediante el uso de operadores, sobreviviendo así los mejores inidividuos. Estos individuos están formados conceptualmente por cromosomas. A continuación, se detalla este funcionamiento:

- Selección: Se crea una población inicial aleatoria, o bien se selecciona una. En ambos casos, se debe asegurar que hay diversidad entre los individuos de la población, es decir, que sean suficientemente distintos los unos de los otros.
- Evaluación: A continuación, se evalúan a los individuos, sabiendo así cuán bueno estos son, en otras palabras, cuán buena es la solución que este nos proporciona.
- Selección: Tras ello, se seleccionan unos individuos, a los que llamaremos "padres".
- Cruce: Los padres producen descendencia, o como se suele conocer, se cruzan entre ellos.
- Mutación: Estos hijos son mutados en algunos de sus cromosomas. De esta forma, llegan a alcanzar valores distintos de sus parientes.
- Reemplazo: Se evalúan a los hijos, y si son suficientemente buenos, pasan a formar parte de la población, reemplazando a otros individuos con peor evaluación. De este modo, se mantiene el mismo número de individuos en la población.
- Evaluación de criterio de finalización: Se comprueba si se ha establecido un criterio definido, por el cual, la ejecución del algoritmo debe parar. Si es así (se cumple), se para en este momento, siendo la población actual la final. En caso contrario, se vuelve a la "selección". Este bucle se repite hasta que la condición se cumpla.

Pese a que su funcionamiento sea ese, antes de podamos pasar a la práctica debemos de conocer ciertos aspectos:

- El cruce y la mutación dependen de ciertas probabilidades que debemos establecer. En caso de no haber cruce, no habrá hijos, luego no hay mutación ni reemplazo, y se pasa a evaluar de nuevo el criterio de finalización. En caso de no haber mutación, se procede con el funcionamiento habitual.
- La selección, cruce y mutación carecen de un funcionamiento específico diseñado. De este modo, existen multitud de operadores de cada uno de ellos, cada uno con sus ventajas y desventajas.

En resumen, debemos de configurar tanto los probabilidades, como seleccionar unos operadores de selección, cruce y mutación adecuados. En la figura 23 se muestra su diagrama de flujo teórico.

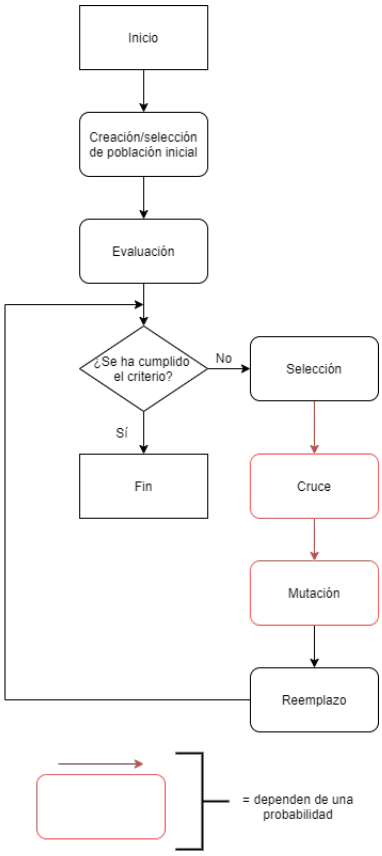


Figura 23: Diagrama de flujo del funcionamiento teórico del algoritmo genético.

3. Objetivo

Nuestro objetivo a cumplir es el de, usando algoritmos de procesamiento de lenguaje natural, llegar a generar secuencias de transformaciones (passes), que intenten mejorar el nivel de optimización de las secuencias ofrecidas por Clang.

Hasta llegar ahí, hemos de plantear cuestiones desde las más básicas, como qué es una frase, cuáles pueden ser útiles para realizar aprendizaje, etc; hasta otras más relacionadas con los algoritmos de lenguaje natural. Todas estas decisiones y cuestiones serán debidamente contestadas y justificadas.

Evidentemente, al ser este un amplio campo de investigación, no se puede cubrir por completo en este documento. La intencionalidad de este es sentar una base del uso de lenguaje natural con la optimización, haciendo uso de varios algoritmos de este tipo y comparando sus resultados con uno de enfoque más clásico.

4. Metodología

4.1. Definición de conceptos básicos y generación de candidatos a frases

Para comenzar, hemos de partir de la presuposición de que existe un lenguaje natural subyacente formado por passes. Sobre este lenguaje subyacente hemos de definir los conceptos básicos de cualquier lenguaje natural.

Una palabra es la unidad de todo lenguaje natural que contiene significado por si misma, separada de otras por un elemento, una breve pausa en el habla y un espacio en la escritura. Asimismo, una frase es un conjunto de palabras con un sentido completo.

Dado que la mínima unidad con significado que tenemos es un pass, al ser este una optimización que realiza un cambio en un programa, lo cual consideramos que equivale a que tenga significado, podemos decir que:

Dado nuestro lenguaje subyacente, un pass equivale a una palabra.

Para la realización de este proyecto se ha tenido en cuenta 87 passes diferentes de los ofrecidos por LLVM, por lo que nuestro lenguaje consta de 87 palabras.

Podríamos pensar que, del mismo modo, un conjunto de transformaciones corresponde a una frase, pero no es así. Necesitamos que ese conjunto de frases tenga un sentido completo, tal y como su definición estipula. Las palabras pueden tener significado individualmente pero carecer de sentido en conjunto, como es evidente.

Dado este problema, ¿cómo nos aseguramos de conseguir frases que tengan un sentido completo? Para ello recurrimos a las secuencias de transformaciones ofrecidas por Clang (O0, O1, O2, O3, OS y OZ) las cuales, al haber sido establecidas como secuencias funcionales para todo programa, han de tener sentido alguno por si misma.

Por este razonamiento, tendríamos tantas frases como secuencias, es decir, 6 en total. Como es evidente, no podemos estudiar el funcionamiento completo de un lenguaje con tan solo 6 frases. Debido a esto, se tomó la decisión de hacer subconjuntos de todas las secuencias unidas una tras de otra, de forma cíclica. Estas subsecuencias son de distinta longitud y comparten elementos entre sí. En vez de tomar una subsecuencia de "n" palabras (desde la palabra 1 hasta la n) y tomar la siguiente a partir de la n+1 palabra, esta siguiente se toma en relación a un valor definido al que llamamos "salto". La siguiente comenzará tras tantas palabras como indique el valor de salto, a partir del inicio de la anterior. A continuación se ilustra con un ejemplo:

Texto original: Mi casa roja tiene un patio enorme muy bonito

Longitud de subsecuencias: 4
Salto: 2

Subsecuencia 1: Mi casa roja tiene
Subsecuencia 2: roja tiene un patio
Subsecuencia 3: un patio enorme muy
Subsecuencia 4: enorme muy bonito **mi**

Como se puede observar en el ejemplo, en la cuarta subsecuencia aparece la palabra "mi". Este suceso es debido a que, como se comentó previamente, se toma de manera cíclica. De esta forma, al no haber suficientes palabras para acabar la cuarta secuencia (la última es "bonito"), se toman de nuevo desde el inicio (empezando con "mi"). Además, este ejemplo ilustra la idea ya mencionada de que no todo conjunto de palabras tiene un significado completo, como se hace evidente al leerlas.

De esta manera, estas subsecuencias son candidatas a ser frases de nuestro lenguaje, para lo cual necesitamos alguna clase de criterio con el que saber si lo son.

4.2. Generación de corpus: criterio relativo al tiempo

Como se ha comentado previamente, ya con la obtención de unas subsecuencias, se ha de comprobar si estas son realmente frases con sentido completo dentro de nuestro lenguaje. Dado cualquier lenguaje natural, esta cuestión se resuelve comprobando si estas tienen o no significado alguno. En nuestro caso no es tan sencillo definir qué tiene "significado".

Se tomó la decisión pues de **establecer un criterio** para responder a este problema, basado en el tiempo de ejecución de un programa estándar, tras aplicar estas transformaciones. Tomamos en cuenta el tiempo de ejecución

debido a que trabajamos siempre con transformaciones que pueden optimizar un programa, en otras palabras, su objetivo es reducir el tiempo de ejecución.

Estas mediciones de tiempo han sido realizadas mediante **perf**, una herramienta de análisis de rendimiento de Linux, ampliamente aceptada y utilizada en tareas de medición como esta, motivo por el cual la hemos usado. Al usarse desde la consola y sin opción de devolver como salida únicamente el tiempo, este se ha obtenido mediante la aplicación de reglas regulares sobre la salida completa de la herramienta. Se indagará más sobre las mediciones de tiempo en el siguiente apartado.

El ya mencionado criterio es el siguiente: si dado un conjunto de transformaciones, las cuales se aplican a un programa, **si el tiempo de ejecución es inferior a un determinado umbral, entonces se considera una frase de nuestro lenguaje**.

De esta manera, el pseudocódigo correspondiente a la generación de las frases sería:

```
generacion_corpus(dimensiones, saltos, tiempo_limite)
    oxs=leer_oxs_unidas()

    para cada dimension en dimensiones:
        para  cada salto en saltos:
            inicio=0
            fin=dimension
            mientras inicio<dimension(oxs):
                subsecuencia=extraer_subsecuencia(oxs,inicio,fin)
                # Se tiene en cuenta que estamos ante un elemento cíclico
                tiempo=medir_tiempo(subsecuencia)
                si tiempo<tiempo_limite:
                    almacenar_en_memoria(subsecuencia, tiempo)
                inicio+=salto
                fin+=salto
```

Tras ejecutar este algoritmo, obtuvimos un corpus inicial de 6180 frases aproximadamente. Dado este valor, podemos decir que tenemos un corpus de escaso tamaño. Téngase en cuenta que, en los problemas de NLP, se trabaja con un gran volumen de frases (considérese el número de frases contenidas en cualquier documento de centenares de páginas, por ejemplo). Se ejecutó con **dimensión desde 10 hasta 500, incluyéndolo, con saltos de 10 palabras**.

4.3. Configuración del benchmark, medición de tiempo

Como hemos visto anteriormente, en el apartado anterior necesitamos medir el tiempo correspondiente a una secuencia de optimizaciones. Para ello, se deben de aplicar estos passes y posteriormente medir el tiempo resultante.

Previamente, en el estado del arte, se comentó de la necesidad de usar un Benchmark. El escogido es **Polybench**, ya que es ampliamente usado en la comunidad científica y académica. Este está formado por una colección de programas en C, lenguaje de programación soportado por LLVM, luego no hay problema en su uso.

Este tiene un aspecto importante a configurar desde su versión 3.0, y es el valor de un parámetro "dataset size". Este indica el tamaño de parámetros que pasan a cada uno de los programas del benchmark. Luego, a mayor tamaño, mayor será la duración del programa resultante. Hay varias opciones, de menor a mayor, son: mini, small, standard, large, very large.

Para ver el tiempo de cada dataset size, se ejecutó cada secuencia de Clang (O0, O1, O2, ...) 100 veces, y se calculó la mediana. El resultado se muestra en la tabla 1.

		Dataset size		
		Mini	Small	Standard
Tiempo (s)	O0	0.01324	0.23717	2011.2590
	O1	0.01035	0.17868	1943.1941
	O2	0.09947	0.16637	1938.5618
	O3	0.00999	0.16638	1940.5421
	OS	0.01146	0.16617	1936.1744
	OZ	0.01146	0.19466	1936.3720

Cuadro 1: Tabla que muestra el tiempo de cada secuencia de optimizaciones de Clang en cada dataset.

De este experimento, sumado a pruebas en otros equipos de menor escala, extrajimos varias conclusiones:

- El tiempo de ejecución depende de factores internos del equipo, tales como el uso actual de la CPU por ejemplo, así como también de los componentes de dicho equipo.
- Con los datasets que producen un menor tiempo de ejecución, el resultado es significativamente variable. Estos son, concretamente, el mini y el small. Para el resto, al tener un tiempo mucho mayor, el resultado varía de manera no significativa.

Debido a que en este problema tenemos un gran volumen de frases de las que debemos medir su tiempo, descartamos el uso de los benchmark standard y superiores. Al tener un tan alto tiempo de ejecución, hace inviable la experimentación, debido a las limitaciones temporales presentes. Un ejemplo de ello, medir el tiempo ejecución de 1000 frases, tendría como resultado una espera de 21 días aproximadamente. De este modo, se escogió el dataset "small", ya que entre ambos, hemos de priorizar el mayor número de pruebas dadas, lo cual nos asegura un tiempo más adecuado, y un tiempo de ejecución menos variable.

De este modo, se estableció que:

El tiempo de ejecución de una frase viene dado por la mediana de los tiempos de 5 ejecuciones, usando el dataset size "small" (Polybench).

El valor de 5 fue escogido debido a que se observó que, con este valor, la desviación estándar con respecto de la media es inferior al 10 %. Este es un consenso aceptado académicamente en el mundo estadístico.

Una vez definido cómo medir el tiempo de ejecución, las funciones usadas para ello son las siguientes:

```
# Generación de un archivo Bitcode (bc) correspondiente a los programas del
# benchmark, para su posterior transformación y medida de tiempo.
#
# Se asegura de crear una copia de seguridad, clean-all, que se mantendrá siempre
# intacta. De esta forma, podemos sobreescribir el archivo all con este.
# Es importante asegurarnos que el archivo Bitcode con el que trabajamos
# no tiene otras transformaciones aplicadas previamente.
def link(route:str, bench:str, data:str):
    # Si el archivo "clean-all.bc" existe, elimina "all.bc", y copia el
    # primero con el nombre del segundo.
    if(os.path.exists(bench+"clean-all.bc")):
        os.system("rm "+bench+"all.bc")
        copyfile(bench+"clean-all.bc",bench+"all.bc")
    # Si no es así...
    else:
        # Se aplica, con el uso de Clang, la parte correspondiente al frontend sobre:
        # los programas del benchmark: preprocesado, análisis, y generación de código
        # en lenguaje IR, codificado en formato bitcode (bc).
        # Como resultado, tenemos un archivo bc de cada archivo c del benchmark.
        os.system(route+"clang-9 -O0 -Xclang -disable-O0-optnone -c -emit-llvm "+bench+"*.c")
        # Movemos esos archivos a la carpeta del benchmark (Clang no los pone ahí).
        os.system("mv "+data+"*.bc "+bench)
        os.system("rm "+bench+"all.bc")
        # Con el uso del comando llvm-link, unimos todos los archivos bc bajo
        # uno solo, llamado "all.bc". Si previamente existe otro, se generará
        # correctamente pero dará error. Debido a esto, en la línea anterior se
        # borra el previamente existente "all.bc".
        os.system(route+"llvm-link "+bench+"*.bc -o="+bench+"all.bc")
        # Creamos nuestra copia limpia, clean-all, a partir del recién creado all.
        copyfile(bench+"all.bc",bench+"clean-all.bc")

# Realizamos transformaciones sobre el archivo previamente conseguido, para
# luego medir y devolver su tiempo, de manera individual, además de su mediana
# y std.
def compile_exec(route:str,bench:str ,passes:str, n, data:str):
    # Realizamos la función link, asegurándonos de que existe el archivo clean-all,
    # y de que el archivo all, sobre el que haremos transformaciones, no tiene otras
    # aplicadas previamente.
    link(route,bench,data)

    # LLVM tiene un fallo. Las transformaciones se pueden aplicar en un solo comando,
    # pero alhacerlo, puede producirse un error y no finalizar la ejecución del comando
    # correspondiente. Se soluciona aplicando las transformaciones una a una.

    # Se intenta aplicar todas las transformaciones en un solo comando.
    result = allinone(route,bench,passes)
```

```
    # Si la aplicación de estas transformaciones no ha tenido éxito (result=False), se aplican
    # uno a uno.
    if not result:
        link(route,bench,data)
        result = onebyone(route,bench,passes)

    # Si tras aplicarlas una a una no ha funcionado, entonces es que la secuencia de
    # transformaciones es incorrecta, hace que el programa no termina. Se le da un tiempo
    # arbitrariamente alto para que se descarte.
    if not result:
        return [1e400,0,[1e400]*n]
    # En caso de que todo haya ido bien, y las transformaciones hayan sido aplicadas a
    # all, se prosigue.
    else:
        # Generamos un archivo objeto, conteniendo código máquina, correspondiente
        # a all.bc, que podemos ejecutar.
        os.system(route+"clang-9 -lm -O0 "+bench+"all.bc -o "+bench+"exec.o")
        # Se ejecuta "runtimes", script que ejecuta n veces perf y devuelve los n
        # resultados.
        cmd = subprocess.check_output("{}runtimes.sh {}exec{}.o {}".format(
                        data,bench,"",n),shell=True)
        # Se procesa y devuelve el resultado.
        times = array(cmd.decode("utf-8")[:-1].split(","),dtype=float)
        average = median(times)
        std_=std(times)
        times=times.tolist()
        return [average,std_,times]

# Aplica las transformaciones con un solo comando y devuelve si ha tenido éxito.
def allinone(route:str ,bench:str ,passes:str) -> bool:
    result = True
    # Con el uso de opt, aplicamos las transformaciones sobre el archivo all.bc
    cmd = subprocess.Popen("{}opt {} {}all{}.bc -o {}all{}.bc".format(
                            route,passes,bench,"",bench,""),
                            shell=True,stdout=subprocess.DEVNULL,
                            stderr=subprocess.DEVNULL)

    # Espera 20 segundos. Si pasan, se termina la ejecución de opt, devolviendo False
    try:
        cmd.wait(timeout=20)
    except:
        if cmd.returncode != 0:
            cmd.kill()
            result = False

    return result

def onebyone(route:str ,bench:str ,passes:str) -> bool:
    result = True
    passeslist = passes.split(' ')

    # Con el uso de opt, aplicamos las transformaciones sobre el archivo all.bc,
    # esta vez una a una.
    for llvm_pass in passeslist:
        cmd = subprocess.Popen("{}opt {} {}all{}.bc -o {}all{}.bc".format(
                            route,llvm_pass,bench,"",bench,""),
                            shell=True,stdout=subprocess.DEVNULL,
                            stderr=subprocess.DEVNULL)
        # Espera 10 segundos. Si pasan, se termina la ejecución de opt, devolviendo False
        try:
            cmd.wait(timeout=10)
        except:
            if cmd.returncode != 0:
                cmd.kill()
                result = False
    return result
```

Ahora, tan solo hace falta un script que lea una a una las frases, se las proporcione a la función "compile_exec", y que guarde los resultados de mediana y desviación por un lado, y los tiempos individuales por otro. El script queda tal que:

```
# Para cada archivo de frases en la variable sentence_files:
for sentences_file_name in sentence_files:
        sentences=[]
        # Se leen y almacenan las frases del archivo actual en la variable sentences.
        with open(pre_route+'frases-generadas/'+sentences_file_name,'r') as file:
            for row in csv.reader(file):
                sentences+=[row]
            file.close()

        # Se realiza la función "link".
        link(llvmpath,basepath,pre_route)

        z=0
        print(sentences_file_name+" begin")

        # Se crea un archivo "result-...-nombre_de_archivo_original" para
        # la mediana y el std.
        with open(pre_route+'result-'+equipo+'-'+sentences_file_name, 'w', newline='') as file:
            # Se crea otro "times-...-nombre_de_archivo_original" para los
            # 5 tiempos conseguidos.
            with open(pre_route+'results/times-'+equipo+'-'+sentences_file_name, 'w', newline='
                ') as file2:
                writer=csv.writer(file)
                writer2=csv.writer(file2)
                # Para cada frase de la variable sentences
                for sentence in sentences:
                    # Se forma una cadena de caractéres de la frase actual, de
                    # forma que pueda ser ejecutado por comandos (pass a pass,
                    # separados por espacios).
                    sentence_=' '.join(sentence)
                    # Se mide el tiempo
                    tiempo,std_,times=compile_exec(llvmpath,basepath,sentence_,runs,pre_route)
                    # Se almacenan la media y desviación estándar en "result"
                    writer.writerow([tiempo]+[std_])
                    # Se almacenan los 5 tiempos en "times".
                    writer2.writerow(times)
                    print(sentences_file_name+" - "+str(z)+"/"+str(len(sentences)))
                    z=z+1
                file.close()
                file2.close()
                print(sentences_file_name+" finished")
```

Estas mediciones de tiempo han sido realizadas en el supercomputador de la UCA, teniendo acceso a él mediante mis directores. Esta elección se debe a que este equipo nos asegura una mínima perturbación a la hora de medir y una mayor estabilidad, en comparación con un equipo tradicional.

4.4. Word2Vec: codificación de palabras en un espacio n-vectorial

Una vez tenemos el corpus, podemos generar la codificación de cada una de las palabras. Para ello, usamos el algoritmo previamente mencionado, Word2Vec.

Como comentamos previamente, el principal problema que esta técnica presenta es la polisemia. En nuestro problema, la polisemia no está presente. Una palabra representa a un pass, y un pass es una operación que siempre realiza el mismo cambio, por lo que podemos decir que siempre tiene el mismo significado.

Con respecto de la arquitectura, se escogió Skip-Gram frente a CBOW, dado que funciona mejor con un conjunto pequeño de datos de entrenamiento.

Tras la arquitectura podemos escoger entre las tres optimizaciones ya explicadas. Mikolov en su investigación [21] realizó una evaluación de estas, cuyo resultado queda reflejado en la tabla 2. Para ello, entrenó distintos modelos de arquitectura Skip-Gram con distintas optimizaciones y analizó su resultado frente a una analogía.

Dado que la precisión total es mayor en el caso del modelo Skip-Gram con negative sampling con k igual a 15, siendo k el número de negative samples, se escogió esta optimización y este parámetro. Esta decisión se ve reforzada por la explicación de negative sampling que Mikolov da, donde recomienda valor de k entre 5 y 20, estando 15 en ese rango. Asimismo, recomienda otro parámetro para la distribución de ruido.

Además de esos valores recomendados, inicialmente se seleccionaron los valores de contexto C=3, y una dimensionalidad, o en otras palabras, longitud de cada word vector de 40.

Method	Time [min]	Syntactic [%]	Semantic [%]	Total accuracy [%]
NEG-5	38	63	54	59
NEG-15	97	63	58	**61**
HS-Huffman	41	53	40	47
NCE-5	38	60	45	53

Cuadro 2: Comparativa de optimizaciones en modelos Skip-Gram, donde NEG-k corresponde a negative sampling con k negative samples, NCE-k a noise constrast estimation con k negative samples, y HS a Hierarchical Softmax

Para su uso, se ha utilizado una librería de código abierto para Python llamada Gensim. Tiene como principal enfoque el modelado de redes de aprendizaje no supervisado y redes usadas en NLP. Concretamente, esta librería contiene una implementación de Word2Vec que una fácil configuración de la red y un fácil entrenamiento.

El código resultante de la creación, configuración y entrenamiento del modelo computacional es:

```
# Se crea el modelo Word2Vec, especificando el tamaño de los word vector (size),
# el tamaño del contexto cogido (window), se especifica la arquitectura usada (sg=1, hs=0)
# y finalmente se indican cuántos negative sampling se toman, así como un parámetro que
# condiciona la distribución de ruido (antes mencionado). min_count refleja cuántas veces
# tiene que aparecer una palabra para que esta se tome en cuenta. Hay más parámetros por
# configurar, los cuales tienen como valor el dado por defecto en la implementación.
model = Word2Vec(min_count=1,size=vec_word_length,workers=cpu_count(), window =window_value, sg
     = 1, hs=0, negative=15, ns_exponent=0.75)

# Se construye el vocabulario a partir de las frases dadas.
model.build_vocab(sent)

# Se entrena el modelo, especificando las frases que se usarán (sentences=...), el número de iteraciones
# que entrenará la red, conocidas como epochs (epochs=...), activamos la opción que muestre el error producido,
# (compute_loss=True), pasamos así mismo una función que procesará el error (callbacks=[MyLossCalculator()]),
# y finalmente especificamos el número de palabras con las que se entranará (total_examples=model.corpus_count).
model.train(sentences=sent,epochs=n_epochs,compute_loss=True, callbacks=[MyLossCalculator()],
    total_examples=model.corpus_count)

# Se guarda el modelo con el nombre especificado en la variable.
model.save("./"+word2vec_file_name)
```

En total, **en nuestro lenguaje tenemos 87 palabras**. Para esta dimensión, **inicialmente se escogió que el tamaño del word vector**, y por tanto la dimensionalidad del espacio en el que se codifica, **sea 40**. En el siguiente apartado, veremos cómo de 87 palabras pasamos a 88, con su consecuente explicación.

La función de Callback usada para visualizar el error durante el entrenamiento es la siguiente:

```
class MyLossCalculator(CallbackAny2Vec):
    # Función de inicialización. Se inicializa el número de epochs y el tiempo del sistema, para
    # tener en cuenta cuántos epochs ha entrenado la red, y cuánto tiempo ha transcurrido.
    def __init__(self):
        self.epoch = 1
        self.previous_epoch_time = time()
    # Esta función se ejecuta al terminar un epoch, una iteración de la red.
    def on_epoch_end(self, model):
        # Se obtiene el error de la red en este epoch
        loss = model.get_latest_training_loss()
        # Se coge el tiempo actual y el anterior para calcular el tiempo transcurrido en este epoch.
        now = time()
        epoch_seconds = now - self.previous_epoch_time
        self.previous_epoch_time = now

        # Se imprimen los datos correspondientes.
        print(f"Loss after epoch {self.epoch}/{n_epochs}: {loss} -> epoch took {round(
            epoch_seconds, 2)} s ")
        # Se actualiza el número de epochs.
        self.epoch += 1
        # Se añade, finalmente, la pérdida a una variable llamada losses2, que posteriormente se puede
        # almacenar en un archivo para visualizar el error a lo largo del entrenamiento.
        losses2.append(float(loss))
        # Se reinicia el error a 0, ya que esta implementación suma el error de los epochs, uno sobre
        # otro.
        model.running_training_loss = 0.0
```

```
        # Finalmente, se podría añadir la línea:
        if(self.epoch%100==0):
            model.save("word2vec-"+str(self.epoch)+".model")
        # Para guardar el modelo cada x epoch. De este modo, podemos guardar entrenamientos
        # intermedios.
```

Este dato mencionado al final de la función de Callback es de **alta utilidad para guardar modelos durante el entrenamiento**. De este modo, nos aseguramos de entrenar una única vez guardando el progreso intermedio del entrenamiento.

4.5. LSTM: Red neuronal recurrente

4.5.1. Creación de modelo, preparación de los datos, entrenamiento

Tras especificar el algoritmo usado en la codificación, únicamente resta especificar el tipo de red usada. Tal y como el título indica, se ha escogido la red LSTM. Esta elección es debida a que en el problema de los passes se encuentra presente la larga dependencia de términos.

Entrando en detalles con nuestro problema, un pass o un grupo de passes puede tener influencia sobre otro presente [11], ya sea anterior o presente más adelante (dato que se deriva del documento citado). No podemos especificar una distancia determinada a la que la influencia de un pass o grupo de passes deja de estar presente sobre otro. De este modo, se escoge a la red LSTM frente a una estándar RNN para solventar este problema.

La tarea de la red es la de, dada una frase, predecir la siguiente palabra a continuación.

La arquitectura de la red usada es la explicada en el estado del arte, de tres capas: una capa de entrada, la capa LSTM, y una de salida. Esta última ejecuta una función de activación softmax para comprobar qué palabra es la más probable según la red. Con respecto de las arquitecturas de RNN, estamos frente a una arquitectura de "many to one", donde se da como entrada una frase, formada por muchas palabras, y se devuelve un único elemento, un vector de probabilidades de cada palabra.

Para ello, se ha usado una librería de Python llamada Keras, de código abierto. Se enfoca en las redes neuronales, ofreciendo la fácil creación redes neuronales personalizadas, con cuantas capas deseemos, del tipo que queramos. Cada una de ellas se puede configurar mediante el uso de parámetros. Ofrece, además, documentación con ejemplos de uso en función de nuestro objetivo, lo cual resulta de alta utilidad de cara al programador. Se ejecuta sobre TensorFlow, otra librería de aprendizaje automático.

La entrada de la red es una frase en codificación index-based. Ya que la entrada tiene que ser numérica, se usa esta codificación en lugar de la de Word2Vec directamente. A la capa de entrada se le proporciona los pesos del modelo Word2Vec correspondiente a su capa de entrada, en otras palabras, a la codificación de todas las palabras. De esta manera, la capa de entrada se encarga de dar a la siguiente capa, la LSTM, la codificación correspondiente por cada palabra de la frase de entrada.

Tras ello, tenemos la capa LSTM. De esta capa únicamente se devuelve un output, como se evidencia al tener una arquitectura many to one. En otras palabras, obtenemos como salida un vector. Sobre este vector se aplica la función softmax, obteniendo, como ya se explicó, un vector tan largo como palabras tengamos en el vocabulario, y en cada posición la probabilidad individual de ser la siguiente palabra que sigue a la frase.

El código referente a la creación de la red es:

```
# Se crea un modelo vacío con la API Sequential (secuencial) de Keras,
# que nos permite tener una pila de capas.
model = Sequential()
# Se añade la capa de entrada, pasándole los pesos preentrenados de Word2Vec,
# es decir, la codificación. "input_dim" corresponde al tamaño del vocabulario,
# y "output_dim" al tamaño de la codificación (de los word vector en nuestro caso)
model.add(Embedding(input_dim=vocab_size, output_dim=emdedding_size, weights=[
    pretrained_weights], mask_zero=True))
# Se añade la capa LSTM, donde "units" refleja el tamaño del vector de salida
# (siendo la salida un vector de tamaño word vector), y "return_sequences"
# un parámetro que controla si la arquitectura es one/many to one, cuando se usa el
# valor False, y one/many to many, cuando se usa True
model.add(LSTM(units=emdedding_size,return_sequences=False))

# Finalmente se le añade una capa tradicional de red neuronal (llamada Dense),
# que tiene como activación "softmax", y donde "units" refleja el tamaño del
# vector de salida.
model.add(Dense(units=vocab_size,activation='softmax'))
```

```
# Se necesitan dos parámetros para hacer funcionar al modelo. Uno de ellos es el optimizer,
# un algoritmo usado para cambiar los atributos de la red, como el peso y el learning rate (o paso).
# El otro parámetro muestra cómo se mide la pérdida o error en nuestra red. Al tener una salida
# probabilística (por softmax), se usa un métrica de error probabilística, llamada
# sparse categorical crossentropy. Calcula la entropía cruzada entre la salida deseada y la predicción.
# Ambos son recomendados por Keras en sus ejemplos para este tipo de problemas
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')

# Estos métodos son de visualización de las entradas y la red. El primero lo muestra en
# consola, y el segundo requiere una imagen pero requiere de la instalación de la librería
# graphviz.
model.summary()
plot_model(model,to_file='model.png',show_shapes=True,show_layer_names=True)

# Se entrena al modelo pasando los datos de entrenamiento, se indica el tamaño del batch
# (número de datos a procesar antes de cada adaptación), y el número de epochs (iteraciones)
# de la red. En cada epoch de Keras se procesan todos los datos de entrenamiento dados,
# actualizando parámetros cada batch.
# La X serían nuestras frases, y la Y sería la palabra a continuación de cada una de las frases de X.
model.fit(train_x, train_y, batch_size=batch_size, epochs=n_epochs_lstm)

# Finalmente se guarda el modelo.
model.save('./'+lstm_file_name)
```

Hay un detalle de la implementación, relacionado con las redes neuronales en general, que es muy importante conocer.

En la generación del corpus, generamos frases de múltiples tamaños. Esto no es problema para Word2Vec, debido a que generaba sus propias entradas y salidas en función del contexto. Sin embargo con LSTM sí tenemos un problema. Como hemos especificado, la entrada de la red es una frase. Por otro lado, la red neuronal requiere como entrada elementos del mismo tamaño siempre. Al tener frases de tamaño variable pero la obligatoriedad de tener un vector codificado de tamaño fijo, se hace evidente la presencia de un problema.

La solución dada es la de tener una palabra nula, sin significado, que usar como "relleno", y hacer que todas las frases sean del mismo tamaño. Este tamaño viene dado por la longitud de la frase más larga presente en el corpus. Es decir, por ejemplo, si tenemos una frase de 10 palabras y la frase más larga tiene 470 palabras, debemos de crear un vector con las 10 palabras, y 460 elementos nulos.

Afortunadamente, Keras tiene como opción la presencia de un elemento nulo, el cual no se toma en cuenta durante el entrenamiento en toda la red. Este elemento nulo corresponde al valor 0 en la codificación, y al valor [PAD] como palabra literal, palabra tomada como estándar general de los problemas NLP. Si nos fijamos en la capa de entrada, se activa la opción "mask_zero=True", que indica que el 0 es un elemento nulo a no tener en cuenta.

De este modo, en nuestra codificación index based, debemos de tener este cambio presente. Esta codificación asigna valores desde 0 hasta n a cada palabra, por lo que el valor "0" es el de una palabra, y a la vez, el del elemento nulo. La solución es simple: sumar 1 a la codificación de cada palabra. De este modo, el 0 se deja en exclusiva para este elemento nulo.

Este cambio también ha de reflejarse en la red, en el parámetro "vocab_size" usado, que reflejaba el tamaño del vocabulario. Ha de sumarse uno, debido a que dentro de nuestro vocabulario tenemos una palabra más.

El código correspondiente previo a la creación del modelo, por lo tanto, sería el siguiente:

```
# Se obtienen los pesos (syn0) del modelo Word2Vec (model_w2v).
pretrained_weights=model_w2v.wv.syn0

# Se obtienen los parámetros de tamaño del vocabulario (las filas de la matriz de peso), y el tamaño
# de la codificación (número de columnas de la matriz).
vocab_size, emdedding_size = pretrained_weights.shape

# Se inserta en la matriz de peso una fila vacía al inicio, correspondiente al elemento nulo.
pretrained_weights=np.insert(pretrained_weights,0, np.zeros((emdedding_size)),axis=0)

# Se suma 1 al tamaño del vocabulario
vocab_size=vocab_size+1

# Se crea una matriz train_x (datos de entrenamiento X) con tantas filas como frases, y columnas
# como la longitud de la frase más larga de nuestro corpus (max_len_sentence).
train_x = np.zeros([len(sentences), max_len_sentence], dtype=np.int32)

# Se crea un vector train_y con tantas filas como frases.
train_y = np.zeros([len(sentences)], dtype=np.int32)
```

```
# Se asignan los valores correspondientes.
for i, sentence in enumerate(sentences):
    for t, word in enumerate(sentence[:-1]):
        train_x[i, t] = word2idx(word)  # La función word2idx devuelve el index de cada palabra
    train_y[i] = word2idx(sentence[-1]) # El index de la última palabra de la frase equivale a la y.
```

Es importante aclarar que, como se ha visto en el código, se usa una función para obtener la codificación de cada palabra. Esta hace uso del modelo Word2Vec, debido a que este ya crea una codificación para cada palabra. Del mismo modo, hay otra función para obtener cada palabra en función del index. Esta última se usa para la generación de texto.

Las funciones correspondientes son:

```
# Palabra a codificación
def word2idx(word):
    if(word=='[PAD]'):  # Si la palabra es PAD, se devuelve 0
        return 0
    else:               # En caso contrario, se devuelve la dada por la codificación de Word2Vec +1
        return (model_w2v.wv.vocab[word].index)+1

def idx2word(idx):
    if(idx==0):          Si la palabra es 0, se devuelve PAD
        return '[PAD]'
    else:               # En caso contrario, se devuelve la dada por la codificación de Word2Vec.
                        # Se debe de restar uno, ya que la codificación de Word2Vec no ha cambiado.
        return model_w2v.wv.index2word[idx-1]
```

Por último, es muy importante saber que si se carga el modelo de Word2Vec, y este ha usado una clase de Callback durante el entrenamiento, como es el caso, esta clase debe de estar presente, pese a que no tenga los mismos elementos. Por ejemplo, se puede usar:

```
class MyLossCalculator(CallbackAny2Vec):
    def __init__(self):
        self.epoch = 1
```

4.5.2. Modelos intermedios en el entrenamiento para estudio de epochs

Keras presenta una particularidad frente al entrenamiento. Y es que, tras ejecutarse una vez la función de entrenamiento, puede volverse a ejecutar prosiguiendo con el entrenamiento como si este no se hubiese finalizado. De este modo, podemos fácilmente hacer un script que entrene un modelo hasta n epochs, pero guardando modelos intermedios cada x epochs.

El código resultante sería el siguiente:

```
epochs_actuales=0
incremento=100
n_epochs=1000

while epochs_actuales<=n_epochs:
    model.fit(train_x, train_y, batch_size=batch_size, epochs=incremento)
    epochs_actuales+=incremento
    model.save('./model_lstm_epochs_'+str(epochs_actuales)+'.model')
```

De esta manera, podemos realizar la generación de frases para cada uno de los modelos, compararlos, y escoger el número óptimo de epochs que debe entrenar la red LSTM.

Podemos sustituir, en nuestro script que entrena, la línea que entrena el número total de epochs por este bucle, de forma que se consigue el mismo resultado, además de los modelos intermedios.

Este comportamiento de guardar modelos intermedios también se puede replicar en Word2Vec, como se observa en el correspondiente apartado de Word2Vec, concretamente, en el código explicativo de la función Callback.

4.6. LSTM bidireccional

El cambio a realizar para poder entrenar un modelo de LSTM bidireccional es muy sencillo. Para ello, se usa una capa llamada "Bidirectional", cuya entrada es a su vez la capa que queramos hacer que sea bidireccional, en nuestro caso, LSTM. De esta forma, con respecto al código anterior de entrenamiento de una LSTM, queda tal que:

```
# Antes, LSTM estándar:
model.add(LSTM(units=emdedding_size ,return_sequences=False))
# Ahora, LSTM bidireccional:
model.add(Bidirectional(LSTM(units=emdedding_size ,return_sequences=False)))
```

Una vez hecho este cambio (cambiar la primera línea por la segunda), el resultado será un modelo de LSTM bidireccional.

4.7. Generación de frases

La generación de texto es sencilla de implementar desde el punto de vista del programador. Existe una función en Keras que, para un modelo entrenado, dada una entrada, devuelve la salida correspondiente. A esta función, por su propio nombre, se lo conoce como de predicción. Con respecto a nuestro problema, la entrada de la función de predicción sería una frase codificada, y su salida sería un índice correspondiente a la palabra siguiente. Esta se añade al final de la frase, de forma que podemos hacer crecer la frase añadiendo palabras al final.

A la frase que se da como entrada se la conoce como "semilla" o "seed" (ya que a partir de esta "crecen el resto"). Es de vital importancia la elección de una buena seed para la obtención de una buena frase, en otras palabras, la obtención de una frase con buen tiempo. Priorizamos las seed de pequeño tamaño con buen tiempo, para que el modelo pueda generar un gran número de frases a partir de ella (consideramos tener una frase distinta por cada palabra que se le añade). Las haremos crecer hasta un tamaño fijado.

Es de vital importancia asegurarnos de que la palabra nula "[PAD]" no llegue a ser interpretada por LLVM como un pass, ya que al no ser un pass de LLVM, sino una palabra creada, obtendremos un error. Para ello, como veremos más adelante, existe un bucle en el que se predice la siguiente palabra hasta que esta sea distinta de "[PAD]". De este modo, si siempre obtuviésemos la misma palabra de la misma frase, estaríamos en un bucle infinito. Sin embargo, tanto la predicción como la elección de cuál es la siguiente palabra dado un vector de probabilidades, conllevan cierta aleatoriedad en su resultado.

Al proceso de escoger la palabra a continuación dado el vector se le conoce como **sampling**. Existen multitud de enfoques para este proceso. Algunos de ellos son:

- Algoritmo voraz: Algoritmo empleado en problemas de optimización, que se asegura de devolver una solución, aunque no sea la más óptima posible. En este, se tiene una lista de candidatos a formar la solución. Se escoge, cada vez, al mejor candidato de la lista y se añade a la solución si este puede formar parte de ella (si es factible). Se repite este proceso de elección y añadir hasta que se haya formado una solución como tal. El pseudocódigo correspondiente sería:

```
devorador(candidatos)-> solucion
    solucion=inicializar_vacio()
    mientras !es_solucion(solucion) y !vacio(candidatos)
        p=seleccion(candidatos)
        candidatos-=p
        si factible(solucion,p)
            solucion+=p
```

- Búsqueda en haz: En este algoritmo se busca expandir un grafo. Para cada nivel de grafo, se escogen los k mejores nodos (nodos con mejor evaluación). Para cada k nodo, se seleccionan otros k nodos del siguiente nivel. Así hasta un número de niveles determinado. Se devuelve la mejor solución, siendo la mejor solución el camino de nodos que tenga la mejor suma total de evaluación de entre todos los nodos seleccionados.

  Cada nodo, evidentemente, tiene su evaluación. En cada nivel, y como hijos de un nodo, están disponibles todos los candidatos a solución. En la figura 24 se muestra un ejemplo.

- Random sampling: Se escogen tantos individuos (individuos, o candidatos a solución, ...) como la solución requiera, siguiendo una distribución binomial. La probabilidad de cada individuo de ser escogido viene dado por una probabilidad (la que se refleja en el vector que obtenemos como salida de la red). Para controlar la aleatoriedad de la elección de individuos, se utiliza una variable llamada temperatura. Esta es usada en múltiples algoritmos. A mayor temperatura, más variable será la solución (será más aleatoria en este caso). A menor temperatura, menos variable. En el caso de ser 0, no será aleatoria y se escogerá siempre la misma solución para el mismo vector dado.

Debido a que solo necesitamos un valor para la solución, con cierto nivel de aleatoriedad implícito, escogimos el último como método de sampling. El código correspondiente a la generación de frases es el siguiente:

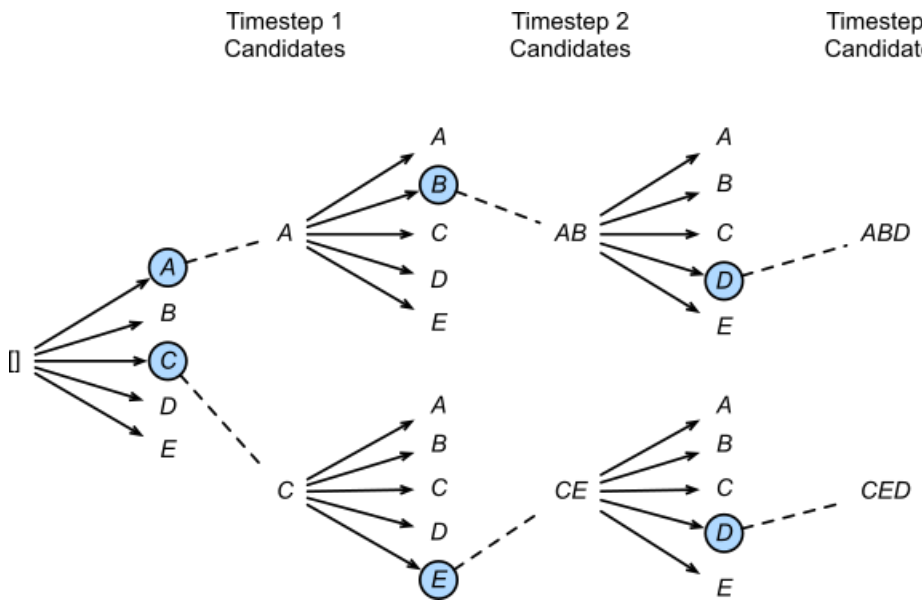


Figura 24: Búsqueda en haz. [2]

```
# Función de sampling: random sampling
def sample(preds, temperature=1.0):
    # Si la temperatura es 0, por lo que no hay aleatoriedad, se devuelve el índice del vector
    # (correspondiente a la codificación) de mayor valor (mayor probabilidad).
    if temperature <= 0:
        return np.argmax(preds)
    # Se hace una conversión de tipo a float64
    preds = np.asarray(preds).astype('float64')
    # Se aplican los cambios pertinentes a la temperatura sobre el vector recibido
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    # Se extrae un valor según la distribución binomial, dado el vector con el cambio de temperatura
    probas = np.random.multinomial(1, preds, 1)
    # Se devuelve el valor correspondiente. Se usa "argmax" debido a que la salida anterior es un
    # vector de ceros con un 1 en el índice del valor escogido).
    return np.argmax(probas)

# Se leen las semillas
with open(pre_route+seeds_file_name,'r') as file:
    for row in csv.reader(file):
        seeds+=[row]
    file.close()

# Se codifican las semillas con la función word2idx (palabra a codificación)
for j,seed in enumerate(seeds):
    for word in seed:
        seeds[j] = [word2idx(word) for word in seed]

# Se crea "final", la variable donde se almacenan todas las frases generadas.
final=[]
# Condición de salida: mientras haya semillas
while(seeds!=[]):
    new_seeds=[]
    # Para todas las semillas en la variable "seeds"
    for seed in seeds:
        word_idxs=seed[:]
        idx=0
        # Mientras la palabra generada sea 0, "[PAD]", sigue generando
        while idx==0:
            # Se obtiene la salida de la red, el vector de probabilidades
            prediction = model.predict(x=np.array(word_idxs))
            # Se utiliza la función sample para la palabra a añadir (codificada).
```

```
            idx=sample(prediction[-1],temperature=temperature_prediction)
        # Se añade la palabra generada
        word_idxs+=[idx]

        # Si la frase con la palabra generada no excede el tamaño máximo que deben
        # tener las frases generadas, se añade al conjunto de semillas que seguirán creciendo
        if(len(word_idxs)<max_len_generated):
            new_seeds+=[word_idxs]

        # Se guarda la frase actual generada decodificada, para su posterior almacenamiento
        final+=[[idx2word(idx) for idx in word_idxs]]
    # Se actualizan las semillas con las que no han excedido el tamaño máximo
    seeds=new_seeds

# Se almacenan todas las frases generadas
with open('./phrases.csv', 'w', newline='') as file:
    writer=csv.writer(file)
    for sentence in final:
        writer.writerow(sentence)
    file.close()
```

Cabe destacar que, el valor escogido inicialmente para la generación de texto, en otras palabras, **el valor de la temperatura, es de 0'7**.

4.8. Coletillas: elemento a añadir en la generación de frases

A lo largo del desarrollo de las diferentes partes de las redes, por experiencia de uso, notamos que en las secuencias de Clang (O0, -O1, ...), a excepción de -O0 debido a su tamaño, se repiten una serie de passes tanto al inicio como al final. Como es lo que "sigue al final de la secuencia", se les decidió llamar "coletilla", o "coleta". Para cuantificar estas coletillas, se implementó un script que recorre desde inicio a final, y de final a inicio, todas las secuencias de Clang mencionadas, contando el número de passes que son iguales conforme se recorren.

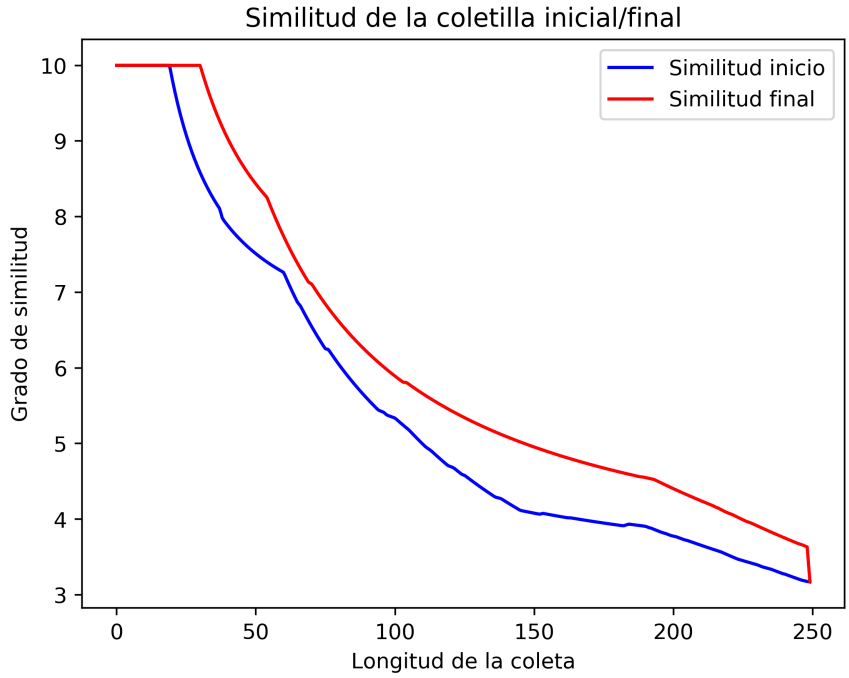


Figura 25: Número de elementos en común de Clang (eje Y) conforme estas se recorren desde inicio a final y viceversa (eje X)

Como se puede observar en la figura 25, todas las secuencias tienen un número significativo de passes que se repiten, como se puede observar en línea horizontal superior, donde la "similitud", número de elementos en común,

es alta. Concretamente, los primeros 20 passes y los 31 últimos de las secuencias son iguales.

De este modo, se les asigna los 20 y 31 passes a las coletillas de inicio y final respectivamente. Pensamos que estas podrían ser un buen añadido a la hora de generar las frases, de forma que todas ellas tuviesen al comenzar la coletilla de inicio, seguido de la frase en sí, y tras ello la coletilla final.

El algoritmo de generación de frases modificado para el uso de coletillas, además de la generación de frases para múltiples modelos, queda tal que:

```
# Se crea la variable coletilla donde se almacenan. Tras ello, se leen del archivo correspondiente.
coletillas=[]
with open("./coletilla-inicial-final-ox.csv",'r') as file:
    for row in csv.reader(file):
        coletillas+=[row]                # 0 corresponde a la inicial, 1 a la final

# Para cada modelo en la variable model_files:
for model_name in model_files:
    # Se lee el modelo
    model=load_model(pre_route+"modelos/"+model_name)
    # Para i desde 0 hasta 2, sin incluirlo. Si i=0, generamos frases sin coletillas.
    # Si i=1, las incluimos.
    for i in range(0,2):
        # Se leen las semillas.
        seeds=[]
        with open(pre_route+seeds_file_name,'r') as file:
            for row in csv.reader(file):
                if(i==0):    # i=0, luego se leen las semillas.
                    seeds+=[row]
                if(i==1):    # i=1, se leen las semillas y se añade la coletilla inicial al inicio
                             # de las semillas.
                    seeds+=[coletillas[0]+row]
            file.close()
        # Se codifican.
        for j,seed in enumerate(seeds):
            for word in seed:
                seeds[j] = [word2idx(word) for word in seed]

        final=[]
        # Se generan frases mientras haya semillas en la variable seed
        while(seeds!=[]):
            new_seeds=[]
            for seed in seeds:
                # Se procede a predecir y hacer sampling como se comentó previamente
                word_idxs=seed[:]
                idx=0
                while idx==0:
                    prediction = model.predict(x=np.array(word_idxs))
                    idx=sample(prediction[-1],temperature=temperature_prediction)
                word_idxs+=[idx]

                # Si i=0, se procede tal y como antes.
                if(i==0):
                    if(len(word_idxs)<max_len_generated):
                        new_seeds+=[word_idxs]

                    final+=[[idx2word(idx) for idx in word_idxs]]
                # Si i=1, se cuenta si la dimensión de la frase actual no excede el máximo,
                # sumando a su longitud la longitud de la coletilla final. Si no es así,
                # esa frase se añade como semilla para que siga creciendo.
                if(i==1):
                    if((len(word_idxs)+len(coletillas[1]))<max_len_generated):
                        new_seeds+=[word_idxs]
                    # Se añade la frase actual generada a la variable final, contenedora
                    # de todas las frases generadas, añadiendo tras la frase la coletilla final.
                    final+=[[idx2word(idx) for idx in word_idxs]+coletillas[1]]

            seeds=new_seeds
        # Si i=0, se guarda en un archivo con el nombre 'new phases' + el nombre del modelo actual
        # + la hora del sistema.
        if(i==0):
            with open('./new-phrases-model-'+model_name[6:-6]+'-'+str(datetime.datetime.now())+
```

```
                '.csv', 'w', newline='') as
                writer=csv.writer(file)
                for sentence in final:
                    writer.writerow(sentence)
                file.close()
            print(model_name+' file generated')
        # Si i=1, se guarda en un archivo con el nombre 'new phases coletilla' + el nombre del modelo
        # actual+ la hora del sistema.
        if(i==1):
            with open('./new-phrases-coletilla-model-'+model_name[6:-6]+'-'+str(datetime.
                datetime.now())+'.csv', 'w', newline='') as file:
                writer=csv.writer(file)
                writer=csv.writer(file)
                for sentence in final:
                    writer.writerow(sentence)
                file.close()
            print(model_name+' + coletilla file generated')
```

4.9. Algoritmo genético

Para la implementación del algoritmo genético, se ha usado un framework llamado jMetal [5], cuyo nombre viene de "**J**ava **Meta**uristic **Al**gorithm" (algoritmos basado en metaurísticas en Java), entre los que se encuentra el algoritmo genético. Afortunadamente, este cuenta con una implementación en el idioma que estamos usando, Python. A esta implementación se le conoce como "jMetalPy".

Este framework hace uso de la programación orientada a objetos, modelando todos los algoritmos y elementos necesarios en clases. Entre estas, destacan las clases de operadores, las de los algoritmos en sí, unas llamadas Solución y otras llamadas Problemas.

Estas últimas clases, Problemas, sirven para modelar a la población a tratar y todo lo referente a ella: generación de población inicial, generación de un individuo de la población, evaluación de un individuo, rango de valores mínimo y máximo de los individuos, etc.

Cada individuo de la población está modelado en una clase Solución, que representa los valores del individuo en una variable llamada "variables", y la evaluación del individuo, que se almacena en una variable llamada "objectives", entre otros datos que almacena esta clase.

Del mismo modo, como se hace evidente, el algoritmo genético está modelado en otra clase llamada "GeneticAlgorithm" donde se define el comportamiento general que este tiene.

Es importante aclarar que en este algoritmo no será necesaria la codificación de word vectors, ya que la codificación en vectores no supone ventaja alguna para este algoritmo, al no tener que entrenar ni aprender, como era el caso de la red. Solo se aplican operadores sobre los individuos, que serán frases. Ya que este algoritmo modifica los valores de los individuos mediante el uso de cruce y mutación, tendríamos vectores con valores distintos de los que partimos, que no sabríamos a qué palabra corresponde, o en la mayoría de los casos, no correspondería con ninguna codificación de ninguna palabra.

Para evitar inconvenientes, y hacer más sencilla la aplicación de operadores, se usa la codificación basada en index (index based) creada previamente por el algoritmo Word2Vec, como hemos mencionado antes. De este modo, además de para evitar estos problemas sin perder beneficio alguno, las operaciones a aplicar por el algoritmo genético son mucho más ligeras computacionalmente (pasando de tener un vector para una palabra, a un entero por palabra).

Sabiendo el problema al cual nos enfrentamos, y teniendo esto en cuenta, se considera que las clases "IntegerProblem", "IntegerSolution" y "GeneticAlgorithm" son las que usaremos. La elección de las dos primeras se debe a que en nuestro problema trabajamos con valores enteros como individuos, al ser frases, a su vez formadas por palabras, con codificación index based. De este modo, se adecúa perfectamente al funcionamiento descrito por esta clase.

Igualmente, se ha de modificar el comportamiento para que realicen lo que queremos, generar nuevas frases. Para ello, haremos uso del mecanismo de herencia, característico de la orientación a objetos.

Además, haremos uso de lo que llamaremos un "diccionario", un archivo donde guardaremos los tiempos de las frases ya ejecutadas, para no tener que ejecutarlas de nuevo, y simplemente consultar el tiempo almacenado en este. Este comportamiento, debido al funcionamiento de los operadores, puede suceder a menudo. Del mismo modo, se usará un archivo "población", donde se almacena la población por iteración, para ver cómo esta evoluciona.

El código referente a la creación de nuestra clase Problema es:

```
# Clase FlagsProblem (así llamamos a nuestra clase Problema modificada) que hereda de IntegerProblem
class FlagsProblem(IntegerProblem):
    # Inicialización de la clase
```

```
    def __init__(self,max_len_sentence:int, upper:int, llvmpath:str, basepath:str, datepath:str
        , runs, log):
        super(FlagsProblem, self).__init__()
        # Se definen los valores máximos y mínimos del individuo. El mínimo valor es 0, y el máximo
        # es 88, tantas palabras como hay en nuestro problema. De ambos, hay tantos como marque el
        # valor max_len_sentence, la longitud de la frase más larga.
        self.lower_bound=[0]*max_len_sentence
        self.upper_bound=[upper]*max_len_sentence

        # Cada individuo tiene tantos valores como marque max_len_sentence. Esto es porque, al igual
        # que con la red, todas las frases tienen la misma dimensión, usando 0 como elemento nulo.
        self.number_of_variables=max_len_sentence
        # Tenemos un objetivo, es decir, un solo valor que considerar: el tiempo de ejecución
        self.number_of_objectives=1
        # El objetivo debe de ser minimizado, es decir, reducido.
        self.obj_directions=[self.MINIMIZE]

        # Se indican las variables para la medición de tiempo de las frases.
        self.llvmpath=llvmpath
        self.basepath=basepath
        self.datepath=datepath

        self.runs = runs
        self.log=log

        # Se crea la variable del diccionario
        self.evaluados=dict()
        # Se lee el diccionario indicado por la variable read_dictionary.
        with open(pre_route+read_dictionary,'r') as file:
            for row in csv.reader(file):
                self.evaluados[row[0]]=row[1]
            file.close()

    # Evaluar a un individuo
    def evaluate(self, solution: IntegerSolution) -> IntegerSolution:
        # Se filtra la frase, quitando los posibles elementos nulos que hayan aparecido.
        sentence=[]
        for idx in solution.variables:
            if(idx!=0):
                sentence+=[idx2word(idx)]
        sentence=' '.join(sentence)
        # Si la frase está en el diccionario, se devuelve el tiempo de este. Si no, se mide,
        # se añade al diccionario, y se devuelve.
        if sentence in self.evaluados:
            tiempo=self.evaluados[sentence]
        else:
            tiempo,_,_=compile_exec(self.llvmpath,self.basepath,sentence,self.runs,self.
                datepath)
            self.evaluados[sentence]=tiempo
            write_dictionary_writer.writerow([sentence,tiempo])
        # Se asigna el tiempo al objetivo 0 (el único que hay)
        solution.objectives[0]=tiempo

        return solution

    # Generación de nuevos individuos (método no usado pero es útil)
    def create_solution(self) -> IntegerSolution:
        # Se crea una variable IntegerSolution
        solution=IntegerSolution(self.lower_bound,self.upper_bound,self.number_of_objectives)
        # Se le asignan tantos números aleatorios como indique el número de variables, según
        # el rango entre el valor mínimo y el máximo.
        solution.variables=[randint(self.lower_bound[i],self.upper_bound[i]) for i in range(0,
            self.number_of_variables)]

        return solution

    def get_name(self) -> str:
        return 'Flags generation problem'
```

A continuación, previamente a incluir el código del algoritmo genético, es importante mencionar una inclusión hecha. Esta es una variable flotante llamada "rate_max_solutions". Se ha observado que, en una población, puede

haber un individuo muy bueno, que produzca un tiempo mucho mejor que el resto, tanto él como otros individuos generados a partir de este (con cruce o mutación). De este modo, en ocasiones se da el caso en que la población se llena de estos individuos, acotando así la diversidad de frases producidas. Para evitar este problema, y obtener mejores soluciones, se incluye este valor. "rate_max_solutions" indica qué porcentaje como máximo puede ocupar en la población un individuo. Por ejemplo, si queremos que, como máximo, el 30 % de la población sea un individuo repetido, "rate_max_solutions" debe de tener un valor de 0'3.

En esta comprobación, se tienen en cuenta que pueden aparecer individuos a priori distintos, pero que tras la eliminación de elementos nulos, sean iguales. Por ejemplo, las frases: 0 1 2 3 0 y 0 0 1 2 3 son distintas en cuanto a comparación 1 a 1 de valores, pero al eliminar los 0 quedan iguales, que es lo que realmente importa.

```
# Clase MyGeneticAlgorithm, que hereda de la clase GeneticAlgorithm
class MyGeneticAlgorithm(GeneticAlgorithm):
    # Función para crear las soluciones iniciales, es decir, la población inicial
    def create_initial_solutions(self) -> list:
        # Se crean tantas soluciones, individuos, como marque el tamaño de la población ya inicializado
        population=[IntegerSolution(self.problem.lower_bound,self.problem.upper_bound,self.
            problem.number_of_objectives)
                for _ in range(self.population_size)]

        # Se asigna a cada individuo las frases codificadas, leídas previamente y almacenadas en la
        # variable sentences_
        for i in range(0,len(population)):
            population[i].variables=sentences_[i]

        return population

    # Función que comprueba si el algoritmo ha finalizado o no. En nuestro caso, usamos esta función
    # para guardar el diccionario cada x iteraciones. En este framework no existe valor que almacene
    # las iteraciones que este lleva. Pero sabiendo que cada iteración se generan "n" nuevos
    # valores (siendo n un valor previamente definido), y que estos n son evaluados, sabemos que
    # hay n evaluaciones por iteración. Por lo tanto, sabiendo el valor de evaluaciones totales,
    # valor que sí se almacena, se puede averiguar el número de iteraciones: evaluaciones/n.

    def stopping_condition_is_met(self) -> bool:
        if self.evaluations % 1000 == 0 or self.termination_criterion.is_met:
            with open(pre_route+'dictionary-'+now+'-epoch-'+str(self.evaluations)+'-ga.csv', 'w
                ', newline='') as file:
                writer=csv.writer(file)
                for sentence, time in self.problem.evaluados.items():
                    writer.writerow([sentence,time])
                file.close()

        return self.termination_criterion.is_met

    # Función de reemplazo. Se junta la población y los individuos generados, se ordenan por la
    # evaluación, y se cogen solamente n, siendo n el tamaño de la población. De este modo, la
    # población tiene un tamaño fijo.
    def replacement(self, population: List[S], offspring_population: List[S]) -> List[S]:
        # Se añaden los individuos generados (offspring) a la población
        population.extend(offspring_population)
        # Se ordenan por objetivo, es decir, la evaluación (el tiempo de ejecución)
        population.sort(key=lambda s: s.objectives[0])

        # Se crea una variable "mod population", que contendrá las variables de la población sin 0.
        # Como se explicó previamente, se crea esta porque dos frases pueden ser a priori distintas
        # pero al eliminar los 0, que sean iguales. Por ejemplo, 0 1 2 3 0 y 0 0 1 2 3 son distintas,
        # pero al fin y al cabo, a la hora de ejecutarse, ambas son 1 2 3.

        mod_population=[]
        for i in range(0,len(population)):
            mod_population+=[population[i].variables]

        for i in range(0,len(mod_population)):
            mod_population[i]=list(filter((0).__ne__, mod_population[i]))

        # Se crea el tamaño límite de veces que puede estar un individuo en la población
        limit_size=int(len(population)*rate_max_solutions)
        i=0
        # Se recorren todos los individuos de la población
```

```
        while i!=len(population):
            # Si un individuo i está más del límite de veces...
            if mod_population.count(mod_population[i])>(limit_size):
                # Se elimina a ese individuo i.
                population.pop(i)
                mod_population.pop(i)
            else:
                # En caso contrario, se pasa a evaluar el siguiente individuo.
                i+=1

        # Finalmente, se guarda la población actual, añadiéndola al final del archivo correspondiente.
        with open(pre_route+'population-'+now+'-ga.csv', 'a', newline='') as file:
            writer=csv.writer(file)
            for i in range(0,len(population)):
                writer.writerow([self.evaluations/len(sentences_)]+[population[i].objectives
                    [0]]+population[i].variables)
            file.close()

        # Se devuelve la población acotada por el tamaño que debe tener.
        return population[:self.population_size]
```

Finalmente, solo queda explicar el código correspondiente a la creación de las correspondientes variables. Para la preparación de los datos, train_x y train_y, se realiza el mismo procedimiento que con la red neuronal LSTM.

```
now=str(datetime.datetime.now())

ga_file_name="ga-sentences-"+now+".csv"

read_dictionary='dictionary.csv'

write_dictionary='dictionary-'+now+'.csv'

rate_max_solutions=0.3
prob_mutation=0.1
prob_cross=0.2

# ...

print('Reading database...')

model_w2v=Word2Vec.load(pre_route+word2vec_file_name)
vocab_size=len(model_w2v.wv.vocab)+1

# Lectura de frases de archivo, se codifican usando las funciones word2idx y idx2word, dejándolas
# con el mismo tamaño todas (usando 0).

# El número de frases totales debe de ser par.
if(len(sentences)%2!=0):
    sentences_=sentences_[:-1]

write_dictionary_file=open(pre_route+write_dictionary, 'w', newline='')
write_dictionary_writer=csv.writer(write_dictionary_file)

# Se enlaza el archivo all.bc
link(llvmpath,basepath,datepath)

print('Database read.')


print('Executing GA...')

# Se crea el problema.
problem=FlagsProblem(max_len_sentence,vocab_size-1,llvmpath,basepath,datepath,runs,log)
# vocab_size-1 ya que si hay 88 palabras en total, el límite debe de ser 87 (de 0 a 87).

print('Number iterations='+str(n_iters)) # Número de iteraciones
print(len(sentences_)*n_iters) # Número de evaluaciones

# Se crea el algoritmo genético.
algorithm = MyGeneticAlgorithm(
problem=problem,
```

```
population_size=len(sentences_),
offspring_population_size=len(sentences_),
mutation=IntegerPolynomialMutation(prob_mutation),
crossover=IntegerSBXCrossover(prob_cross),
selection=RouletteWheelSelection(),
termination_criterion=StoppingByEvaluations(max_evaluations=len(sentences_)*n_iters)
)
# Se hacen tantas evaluaciones como offspring_population_size, ya que esos son los individuos
# que se generan del cruce en cada iteración, los cuales son evaluados.

algorithm.run()

print('GA finished.')

print('Saving information...')

# Se almacena las frases de la población final, teniendo en cuenta que estas no estén repetidas, y
# sin que las palabras estén codificadas (se decodifican).
unique_result=[]

with open(pre_route+ga_file_name, 'w', newline='') as file:
    writer=csv.writer(file)
    for solution in algorithm.solutions:
        result_sentence=[]

        for idx in solution.variables:
            word=idx2word(idx)
            if(word!='[PAD]'):
                result_sentence+=[word]
        result_sentence+=[str(solution.objectives)[1:-1]]

        if result_sentence not in unique_result:
            unique_result+=[result_sentence]
            writer.writerow(result_sentence)
    file.close()

write_dictionary_file.close()

print('Saved in "'+pre_route+ga_file_name+'". Program finished.')
```

4.10. Dinámica general para la generación de nuevas frases

A continuación, se incluye un diagrama en el que se resume la dinámica general seguida desde la creación del corpus inicial hasta la generación de frases, usando los modelos mencionados (figura 26):

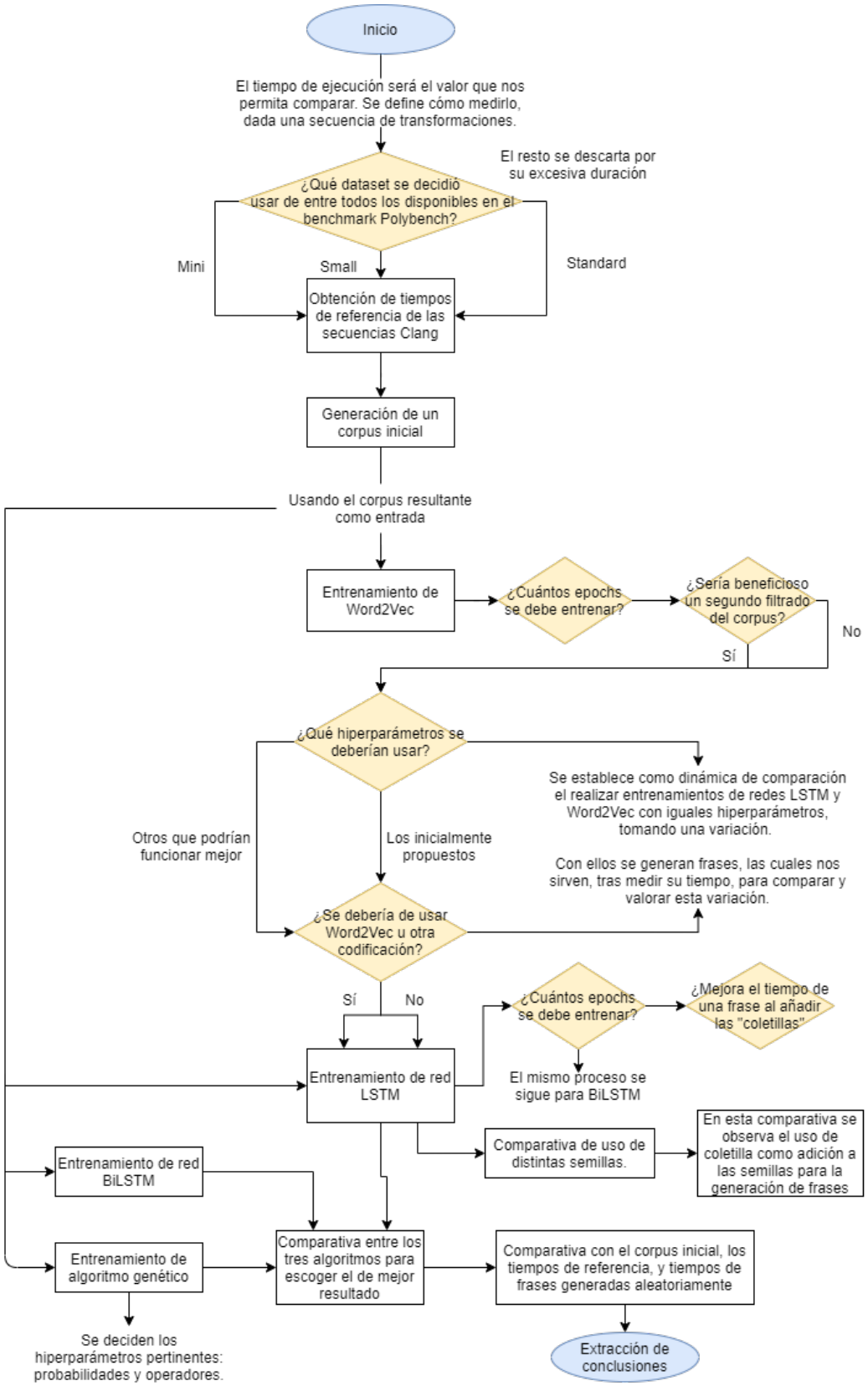


Figura 26: Esquema general que muestra resumidamente los pasos seguidos desde el inicio hasta generar las frases con distintos modelos.

5. Discusión

5.1. Word2Vec

5.1.1. Evaluación del modelo Word2Vec: configuración de parámetros

Para comenzar, hemos de evaluar cuán bueno es nuestro modelo Word2Vec, y probar con distintas configuraciones de sus parámetros. Es aquí donde comienza nuestro primer problema.

Al contrario de lo que podríamos pensar, el error obtenido durante el entrenamiento no nos sirve para evaluar nuestro modelo completamente. Sabemos que el error se calcula teniendo en cuenta la palabra central y su contexto, como se explicó previamente. De este modo, el objetivo es minimizarlo, pero no debe llegar a 0, o en ocasiones, ni siquiera puede.

Recordemos que el objetivo del entrenamiento es, en un espacio n-dimensional, encontrar el lugar de cada palabra de cada vocabulario en este espacio. Los word vector, o en otras palabras, la codificación de cada palabra, son las coordenadas de este punto en ese espacio.

El error es, en definitiva, la distancia entre un punto que representa a la codificación de la palabra central, y los puntos de las palabras de su contexto. Ya que toda palabra, tomada como central, tiene varias a su vez de contexto (el número de palabras tomadas como contexto viene dado por la ventana), la distancia de un punto a varios no puede ser 0. Un punto, la palabra central en este caso, no puede encontrarse a distancia 0 de tantos puntos como palabras de contexto tenga. El objetivo es minimizar ese error, consiguiendo encontrar una ubicación en el espacio para cada palabra, donde la distancia de esa palabra a todas las de su contexto sea mínima.

Respecto a la afirmación de que el error no puede ser 0, podría encontrarse alguna excepción. Por ejemplo, si todas las palabras de contexto son iguales que la central. Aunque en tal caso, en un lenguaje común, esa frase carecería de sentido.

El error no nos sirve para hacer una comparación entre configuraciones de modelos. Supongamos que queremos comprobar qué es mejor, si una ventana de 5 o de 20, dejando el resto de parámetros por igual entre ambas configuraciones. Recordemos que la ventana, durante el entrenamiento, es el número de palabras de contexto que se toman con respecto de una palabra llamada central, que forman las salidas u entradas en función de la arquitectura escogida. Al observar el error cuadrático medio o MSE, estimador del error que mide el promedio de los errores al cuadrado, puede que ambos tengan un error parejo aproximadamente, o que este sea dispar. Independientemente de esto, la codificación resultante se habrá generado con distinta información, por lo que la codificación resultante será dispar entre ambas configuraciones, teniendo diferentes resultados con respecto de nuestro sistema. Como es evidente, nada de esto se refleja en el error.

En conclusión, podemos afirmar que el error no nos sirve para evaluar cuán buena es nuestra codificación resultante, usando una configuración determinada, sobre nuestro sistema final. Sin embargo, sí nos sirve para saber cuándo debemos de finalizar el entrenamiento, observando cómo este se estabiliza. Además, nos permite ver si este se realiza adecuadamente, pudiendo observar si la tendencia general del error es descendiente conforme pasan los epochs.

¿Cómo evaluar pues? **No hay una manera objetivamente correcta de evaluar** la codificación resultante, depende de la aplicación de nuestro sistema (y podemos imaginar que aplicaciones hay demasiadas). Pese a esto, en [26] se recomiendan las siguientes tareas a realizar de manera estándar:

1. Similitud entre pares de palabras: La evaluación consiste en encontrar la similitud coseno (valor del coseno del ángulo comprendido entre ellos) entre pares de palabras “a” y “b”, y más tarde calcular el coeficiente de correlación de Spearman. De esta forma, evaluamos la similitud entre ambas palabras.

2. Analogías: Dada una analogía “a” es a “b” como “c” es a otra palabra, se calcula la precisión con la que se encuentra esta otra palabra restante no dada en la analogía.

3. Categorización: Se basa en agrupar palabras de nuestro vocabulario en categorías o clusters, para más tarde analizar esta agrupación.

Para estas tareas, se utilizan datasets ya preparados, ya sea con pares o analogías. Para nuestro problema, debido a que se está empezando a investigar, no existe dataset hasta el momento para ello. De esta manera, para la evaluación se tendría que usar la notación humana, es decir, realizar alguna de las tareas de forma manual.

El conocimiento que tenemos de nuestras palabras, los passes, es qué realizan cada uno de ellos, y cuáles suelen aparecer junto a otros. Al ser operaciones cuyo objetivo es optimizar, no tiene sentido que hayan operaciones análogas de sentido contrario, cuyo objetivo sea “desoptimizar”, luego la realización de analogías es una difícil labor. Tampoco se han encontrado similitudes de manera manual entre los passes, debido a que se evita que las operaciones

sean redundantes. Incluso si tratan elementos en común, como es el caso de varios passes de transformación que giran en torno a bucles, no realizan los mismos cambios, luego no tienen significados parecidos. Por lo que, de nuevo, la similitud entre palabras es una difícil labor.

Puede que, en el caso de las transformaciones que tratan elementos comunes, puedan encontrarse similitudes. O puede que incluso se puedan hallar por cercanía. En ambos casos, requiere de anotación manual y un profundo estudio de las transformaciones, entre otras cosas. Este método queda fuera del alcance del proyecto.

Sabiendo esto, se ha tomado la decisión de no evaluar la configuración del modelo y la codificación conseguida mediante las tareas recomendadas. En vez de eso, **se considerará la configuración de este modelo como un factor a tener en cuenta a la hora de evaluar la salida generada**, es decir, a la hora de evaluar las frases generadas por el modelo LSTM.

Este hecho no quiere decir que se vayan a escoger valores arbitrarios para los parámetros. Como ya se vio en la metodología, algunos de ellos ya han sido escogidos en función de resultados empíricos y recomendaciones hechas junto a la propuesta de uso de Word2Vec. Otros parámetros serán escogidos según los resultados que se han observado tras ciertos análisis. Concretamente:

- Tamaño de los word vector: Por lo que han concluido algunos experimentos y análisis [26], se dice que existe un cierto "umbral mínimo" para el que la evaluación pasa a ser estable, y por debajo de este el rendimiento es pobre. Como sabemos, no podemos evaluar nuestro modelo, luego no podemos encontrar este umbral. Si fuese así, existirían varios posibles acercamientos como optimización bayesiana, grid search, etc.

  Por otro lado, se sabe que son habituales los valores de 100 y 300. Sin embargo, el valor concreto a usar depende del tamaño de nuestro vocabulario y la tarea a realizar. Dado que nuestro vocabulario solo consta de 87 palabras (sin contar el ya mencionado elemento nulo, como se ve en 4.1 y 4.5.1), podemos asegurar que el tamaño debe ser inferior a 100. De esta manera, **se escogió un valor inicial de 40**. Más tarde se determinó que con un valor de 10 se obtendría un mejor resultado. Se considera como suficiente un espacio de 10 dimensiones para representar los 87 puntos que conforman nuestro vocabulario.

  Este valor tiene gran importancia, ya que si escogemos un valor inadecuado, puede dar lugar a que obtengamos una representación que perjudique a nuestro sistema, llegando a obtener una representación similar a la de one-hot, en el caso de una elección muy inadecuada.

- Tamaño de la ventana: La decisión de este parámetro es de suma importancia, ya que su valor refleja el número de palabras de contexto usado, y por lo tanto, afecta a qué información se refleja en la codificación obtenida. Según se ha concluido tras varios análisis [17], valores grandes de ventana suelen capturar información relacionada de cada palabra, es decir, qué otras palabras se suelen usar alrededor de una. Por otro lado, valores pequeños suelen capturar información de cada palabra en sí, como cuáles otras son similares. En otras palabras, con un valor pequeño se le da importancia a las palabras cercanas, y con un valor grande a estas y a otras de su alrededor, su contexto, a costa de un mayor coste computacional. Esto plantea varias preguntas:

  - Significado de "valor grande" y "valor pequeño": El concepto de grande o pequeño depende de los datos que tengamos, de la dispersión de las palabras en una frase. Según se ha observado, en conjunto de datos de idioma inglés, se tiende a considerar un rango de valores de entre 10 y 1, siendo el primero grande y el segundo pequeño.
  - ¿Cuál es el valor más correcto? Después de pruebas sobre varios datasets sobre el mismo rango de valores antes especificado [18], se señala que para las pruebas de similitud de pares, tamaños pequeños funcionan bien. Sin embargo, para las analogías, en algunos datasets funcionan bien tamaños grandes, y en otros, tamaños pequeños.

    Como resultado de estudiar la dependencia del contexto considerado [17], se concluye que no se puede asignar un significado determinado a cada uno de los valores usados (los que probaron se incluyen dentro del rango), pero sí podemos vislumbrar qué información se captura en la codificación, como se señaló al inicio.

  Dado que nos interesa capturar la información de cada palabra en sí como información relacionada de cada una de ellas, y al no haber un consenso, se decide por escoger un valor intermedio del rango mencionado. **Inicialmente, se ha escogido un valor de 3**, aunque se considera a tomar en cuenta un valor de 5.

5.1.2. Error resultante del entrenamiento Word2Vec

Tras decidir inicialmente los parámetros a usar para Word2Vec, el siguiente paso a dar es el de realizar el entrenamiento. Para ello, se pretendía observar el comportamiento frente a un lenguaje natural común, como es el

inglés, y luego compararlo con el resultado obtenido con nuestras frases.

Para ello, se obtuvo un dataset ya preparado de una implementación de Word2Vec [29]. Tras un entrenamiendo de 1000 epochs, el error cometido es el siguiente (figura 27):

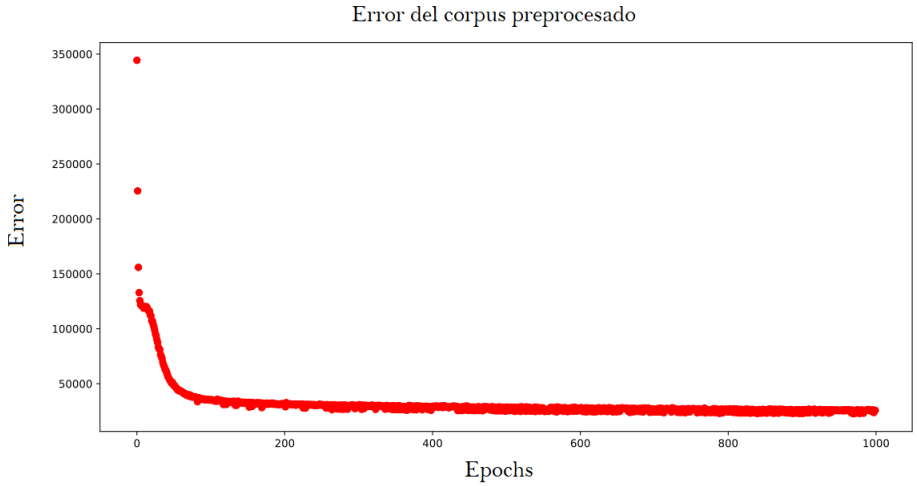


Figura 27: Errores (eje Y) de W2V conforme pasan los epochs (eje X) en un dataset preprocesado.

El resultado es el que podemos ver en multitud de modelos. Al inicio el error es algo, y conforme va continuando el entrenamiento, el modelo aprende y el error baja a estabilizarse, como podemos ver en el epoch 100, aproximadamente. Sin embargo, al amplicar sobre la zona estable a priori, observamos más detalles:

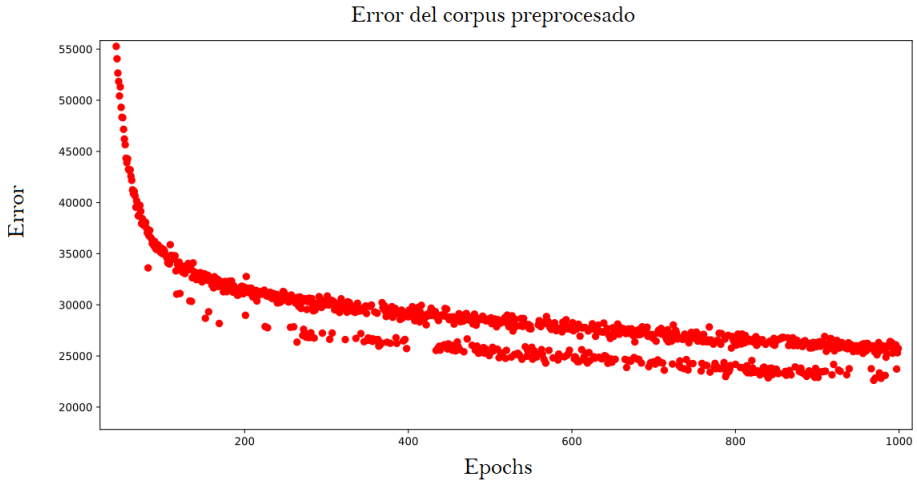


Figura 28: Enfoque sobre algunos errores (eje Y) de W2V conforme pasan los epochs (eje X) en un dataset preprocesado.

Como podemos observar, los errores se bifurcan en dos claras líneas horizontales, una inferior con menos valores, y una superior que sigue la tendencia de la curva.

Por otro lado, el resultado que obtenemos con nuestro corpus generado es el siguiente (figura 29).

Como podemos observar, el error obtenido no forma una curva como claramente ocurría en el primer caso. Uno de los posibles motivos es que no estamos tratando un dataset preparado para el aprendizaje, o quizás a la desigualdad de calidad de los corpus.

Pese a esto, se aprecia que ambos se bifurcan por igual conforme entrenan en varias rectas, una principal con más valores de error y otras menos pobladas. En el caso del corpus anterior era solo una, mientras que este son dos, una superior e inferior.

Existe una notable diferencia entre ambos rangos de errores cometidos, como se puede ver al observar los ejes verticales. Pese a ello, es importante señalar que la tendencia descrita por los valores de error es descendente. Esto

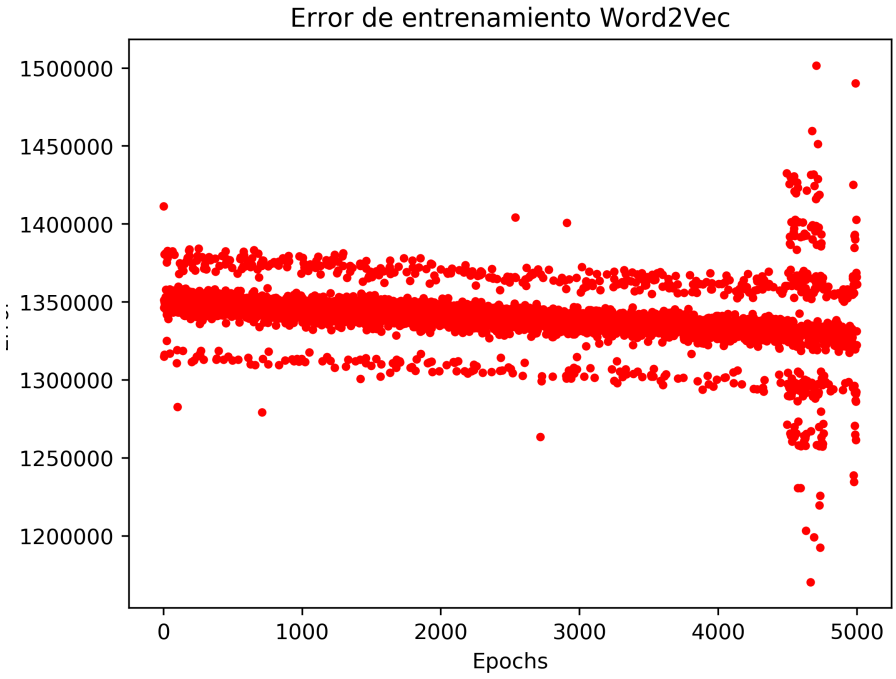


Figura 29: Errores (eje Y) de W2V conforme pasan los epochs (eje X).

nos muestra que, conforme transcurren los epochs, el algoritmo aprende exitosamente y tiende a la convergencia.

5.1.3. Número de epochs de Word2Vec

Una vez tenemos los errores del entrenamiento del modelo, a continuación se decidirá el número de epochs de la red, o como se le puede considerar, el número de iteraciones de ésta.

Este es un parámetro de suma importancia a escoger. Esta relevancia viene a raíz del llamado overfitting o sobreentrenamiento. Por lo general, si dejamos que la red entrene un número demasiado alto de epochs, esta puede adaptarse demasiado al conjunto de entrenamiento. En otras palabras, el error empírico, el producido por los datos de entrenamiento, será muy bajo. Por otro lado, si probamos con otros datos provenientes de otro conjunto diferente, al que usualmente conocemos como de validación, el error correspondiente a este, llamado de generalización, podría ser muy alto.

Para evitar este problema y conseguir un buen valor para ambos tipos de errores, usualmente se aplican los llamados métodos de validación. Cada uno de ellos consiste en entrenar nuestro modelo con distintos tratamientos de los datos disponibles. Tras ello, se observan ambos errores y se decide un valor óptimo de epochs donde ambos errores sean válidos, sin que se haya producido overfitting.

Sin embargo, dada la naturaleza de esta red, nos encontramos en un caso distinto al descrito. Validar es, al fin y al cabo, verificar la calidad del modelo para todo conjunto dado. Para esta finalidad, existen los ya mencionados métodos de evaluación. Tampoco podemos aplicar, como se ha descrito, varios conjuntos de datos, entrenamiento y validación. Esta red trata de encontrar la posición adecuada para todos los puntos (palabras de nuestro vocabulario) en un espacio, usando todos los datos disponibles. Cuando estas palabras están situadas, el error habrá convergido (se encontrarán en la posición adecuada y el error no descenderá más, al no necesitar reposicionarse), y podemos finalizar el entrenamiento. De esta forma, podemos deducir el número de epochs adecuados.

En resumen, determinamos el número de epochs observando el error, ya que este refleja la diferencia entre las salidas deseadas (contexto) y la generada (palabra central), y en cierta manera, cuánto ha de reposicionarse, sin llegar a ser 0.

Podemos observar que a partir del epoch 4500, los valores de error comienzan a dispersarse a lo largo del eje Y. Debido a que no se desea introducir más dispersión del error, pudiendo causar una pérdida de calidad en el resultado del modelo, se decide que el valor de epoch **5000 será el óptimo para el entrenamiento de esta red.**

5.1.4. Criterio de reducción de tamaño del corpus en función del error Word2Vec

Tras entrenar el modelo y observar el error resultante, se pensó que cabía la posibilidad de que todas las frases no fuesen igual de buenas de cara al entrenamiento pese al criterio de tiempo de O0. Por ello, se pensó en eliminar un fragmento del total de las frases, empezando por las de mayor tiempo, y ver el resultado generado. Para ello, se establece un criterio similar al ya establecido, pero reduciendo el tiempo máximo permitido a un porcentaje del tiempo de O0.

En otras palabras, se hace un segundo filtrado del corpus, estableciendo un nuevo criterio de tiempo. Anteriormente, toda frase que formase parte del corpus debía de tener un tiempo inferior al de O0. Con este nuevo criterio, éste debe ser inferior al de un porcentaje del tiempo de O0 (0.9*tiempo(O0), el "90 % de O0, como ejemplo).

El resultado de los distintos errores conforme progresa el entrenamiento, con distintos tiempos con respecto de O0, es el que aparece en la figura 30.

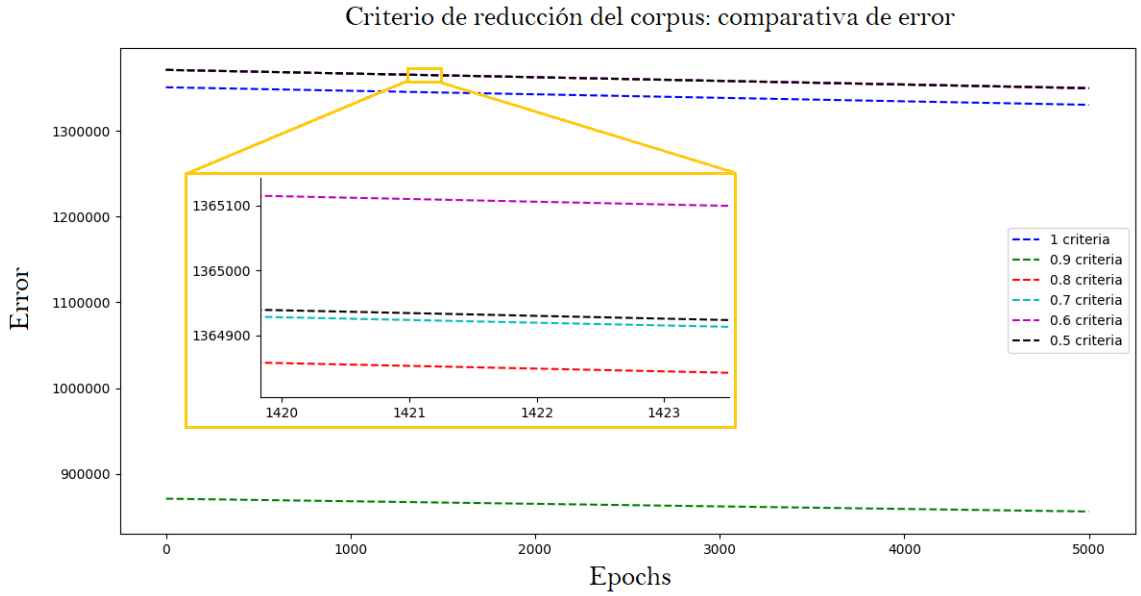


Figura 30: Comparativa del error conforme transcurre el entrenamiento al variar el nuevo criterio temporal. En el rectángulo se aprecian varias líneas de error paralelas.

Como podemos observar, se produce una gran mejoría en el error conseguido al reducir el tiempo máximo al 90 % del original. Este decremento puede deberse debido a que entre esas frases quitadas se encuentre alguna que empeore el modelo de forma significativa. Sin embargo, vemos que en el 80 % en adelante el error cometido vuelve a subir, quedándose tanto este como los restantes alrededor del mismo valor. Del mismo modo, una justificación sería que se ha eliminado alguna frase que era muy beneficiosa para el modelo, y de algún modo, "equilibre la balanza" (se eliminan frases perjudiciales y otras beneficiosas de forma que queda con igual error).

Estas justificaciones son una posible razón la cual no puede ser comprobada, entre otros motivos, debido a las limitaciones temporales impuestas. De todas maneras, una reducción considerable del error no necesariamente implica una mejora en la calidad de nuestra codificación, cosa que no podemos comprobar. De hecho podría incluso empeorar al tener menos frases con las que entrenar, y por lo tanto, aprender menos.

En conclusión, no podemos concluir en si repercute positiva o negativamente en la codificación resultante, debido a la imposibilidad de evaluar. Sin embargo, uno de estas reducciones, la del 90 %, disminuye considerablemente el error producido, lo cual podría ser positivo. Por consiguiente, se decide por usar esta reducción. Como trabajo futuro, queda pendiente un análisis más detallado de esta decisión tomada.

5.1.5. Comparativa de configuraciones de hiperparámetros Word2Vec

Tras la decisión del número de epochs, se desea hacer una comparativa de distintas configuraciones de hiperparámetros de Word2Vec. Esta viene motivada debido a que al inicio se escogieron unos valores iniciales para varios hiperparámetros, y más tarde se consideró que otros valores podían conseguir un mejor resultado. En concreto, los

hiperparámetros tomados en cuenta son los de tamaño de ventana y tamaño de vectores o dimensionalidad. La configuración inicial escogida y la adicional fueron comentadas en el apartado 5.1.1.

Como se mencionó en el mismo apartado, el error es un valor que no nos sirve para comparar configuraciones de modelos. De modo que, como se comentó, la evaluación se haría junto a la red LSTM. En este caso, los parámetros de la red LSTM no están escogidos, ya que la decisión de los de Word2Vec establecen una base desde la que poder continuar. De modo que, a falta de parámetros finales, se escogerán algunos a priori, válidos para hacer la comparación.

Esta dinámica, la de escoger unos valores por defecto para ambas redes, variar en determinadas decisiones, y entrenar para luego generar y observar los resultados en los tiempos generados, **se repetirá a lo largo de los próximos experimentos**. **Se establece como una metodología para observar la influencia de una determinada elección**.

Para realizar la comparativa, entonces, se han realizado dos entrenamientos de dos configuraciones distintas de Word2Vec. Por otro lado, en ambos entrenamientos, los hiperparámetros de la red LSTM se han mantenido iguales.

A modo de repaso, los hiperparámetros usados para este experimento son:

- Configuración 1:
  - Word2Vec:
    - Ventana: 3
    - Tamaño de los vectores (size): 40
- Configuración 2:
  - Word2Vec:
    - Ventana: 5
    - Tamaño de los vectores (size): 10
- Parámetros que se mantienen iguales en ambas configuraciones:
  - Word2Vec:
    - Epochs: 5000
    - Los especificados en 4.4, además de los de por defecto (redactados en la documentación de Gensim)
  - LSTM:
    - Epochs: 300
    - El resto de parámetros tal y como fue especificado previamente en 4.5.1
  - Generación de texto: Random sampling con un valor fijo de temperatura de 0.7
  - Semillas usadas: 8 frases de longitud 10 por debajo del percentil 30 con respecto del corpus ordenado por tiempo de menor a mayor.

A continuación, podemos visualizar los tiempos de ambas configuraciones (figura 31).

Podemos ver cómo claramente ambas configuraciones tienen un rendimiento totalmente contrario uno con respecto del otro. Al inicio, la configuración "New" tiene un mejor resultado, pero empeora conforme aumenta el tamaño de la frase. El mejor resultado que ofrece esta configuración se ve superado por la otra, "Old", cuando alrededor del valor 250 del eje X desciende y pasa a estabilizarse, coincidiendo con el tamaño de las secuencias Clang.

Tiene sentido que, al aumentar el tamaño de las frases (eje X), el tiempo resultante baje. De modo que, al conseguir la configuración "Old" un resultado mejor y estable (se mantiene a lo largo del eje X), decidimos escoger esta configuración.

Se ha vuelto a realizar este experimento, para comprobar si se consigue el mismo resultado. Pese a variar ligeramente, el resultado que podemos extraer es el mismo. A continuación se muestra.

5.1.6. Uso de Word2Vec

Como se puede observar en el apartado de evaluación de Word2Vec, no hemos podido realizar una evaluación de nuestra codificación. Por lo que, en definitiva, no hemos podido comprobar la calidad de la codificación resultante, o variar los parámetros para mejorarla.

Este hecho nos hace cuestionar si nuestra codificación conseguida es beneficiosa o no con respecto a la generación de frases. Es cierto que, como comentamos en el correspondiente apartado de metodología, Word2Vec supone una

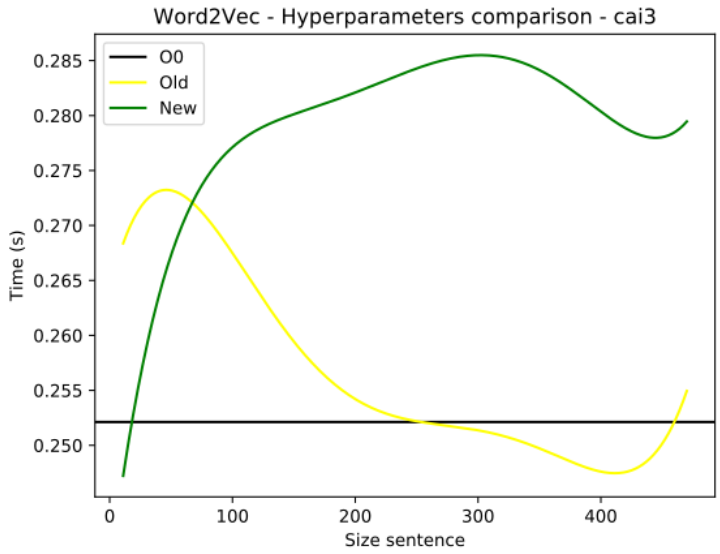


Figura 31: Comparativa de los tiempos conseguidos usando los hiperparámetros "Old" (ventana 3, tamaño 40) y los "New" (venta 5, tamaño 10), y los tiempos de referencia de Clang.

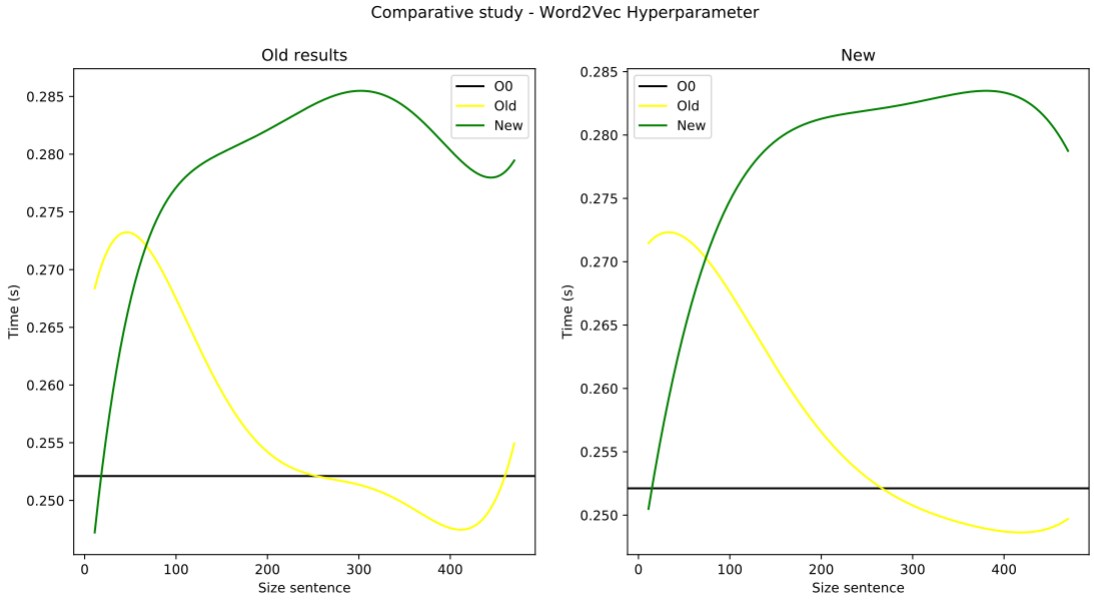


Figura 32: Comparativa entre los dos resultados del experimento de hiperparámetros.

mejora en cuanto a tiempo de computación, y aunque es sustancial, usar otras codificaciones (one hot o index based) supone un coste computacional aceptable. Dado que tampoco podemos garantizar su calidad, puede darse el caso de que otras codificaciones consigan un resultado similar o mejor.

Debido a esta última posibilidad comentada, se hace necesaria la comparativa.

Para ello, se han entrenado dos modelos LSTM con los mismos parámetros, al igual que se hizo previamente. A uno se le ha proporcionado los pesos iniciales correspondientes a la codificación Word2Vec. En el caso del otro, los pesos tienen inicialización aleatoria, de modo que solo se usa la codificación inicial de los datos de entrenamiento (index based, usada también en el otro modelo, vea 4.5.1).

Si se desea ver los parámetros usados, véase el apartado anterior.

Tras entrenar ambos, se generan frases usando las mismas semillas, se miden sus tiempos, generando la gráfica a continuación (figura 33).

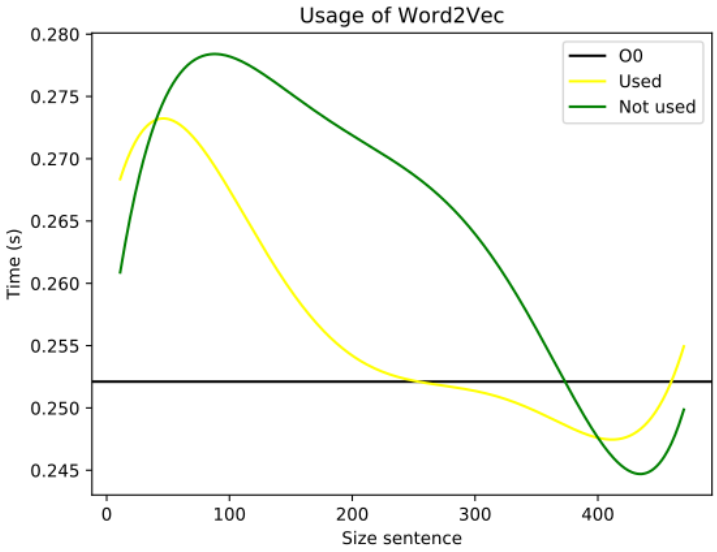


Figura 33: Comparativa entre un modelo LSTM con Word2Vec, y otro sin, además de los tiempos de referencia de Clang.

El resultado de este experimento muestra cómo el uso de Word2Vec produce claramente un mejor resultado temporal. De esta manera, se decide continuar con su uso.

Al igual que anteriormente, se ha vuelto a hacer el experimento para observar si se produce el mismo resultado. Como se puede observar en la figura 34, se ha conseguido exitosamente pese a las leves diferencias producidas.

5.2. LSTM y generación de texto

5.2.1. LSTM: Elección de epochs

Una vez hechas las pruebas pertinentes relacionadas con la codificación, se prosigue con la red neuronal principal, LSTM. En concreto, se comienza a estudiar el número de epochs.

Como ya mencionamos antes, los epochs tienen una gran importancia de cara a la configuración, debido al sobreentrenamiento. Para evitarlo, y escoger este valor correctamente, usamos los métodos de validación. En este caso, al contrario de Word2Vec, podemos aplicar los métodos de validación de forma habitual, comprobando error empírico y de generalización.

El método escogido es el de validación simple. Es tan válido como cualquier otro, y además, debido a la elección de tecnologías y paquetes es el de implementación más sencilla. Consiste en dividir los datos en dos grupos, entrenamiento y validación. Con el primer grupo se entrena, mientras que con el segundo, el cual no ha tenido contacto con la red, se evalúa el modelo conseguido.

Para escoger correctamente el número de epochs, se ha entrenado un modelo LSTM durante 1000 epochs usando este método de validación, guardando los errores producidos en cada epoch, tanto empírico como de validación (producido con los datos de entrenamiento y con los datos de test respectivamente). El objetivo es escoger un rango de epoch exacto en el cual el error sea estable y no sea efectivo continuar con el entrenamiento.

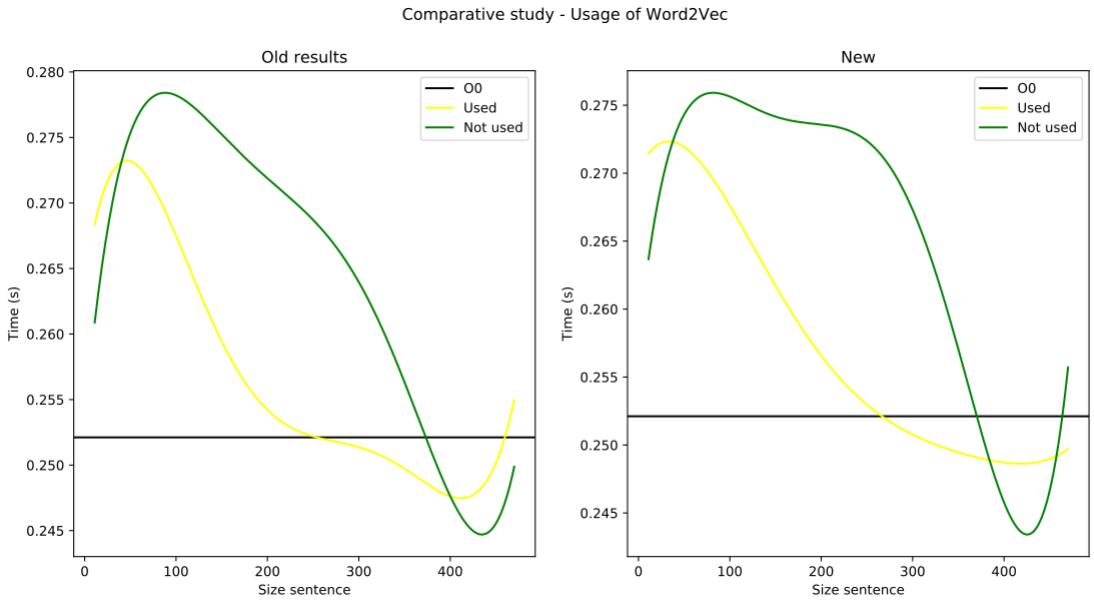


Figura 34: Comparativa entre los dos resultados del experimento de uso.

Los errores se muestran en la figura 35.

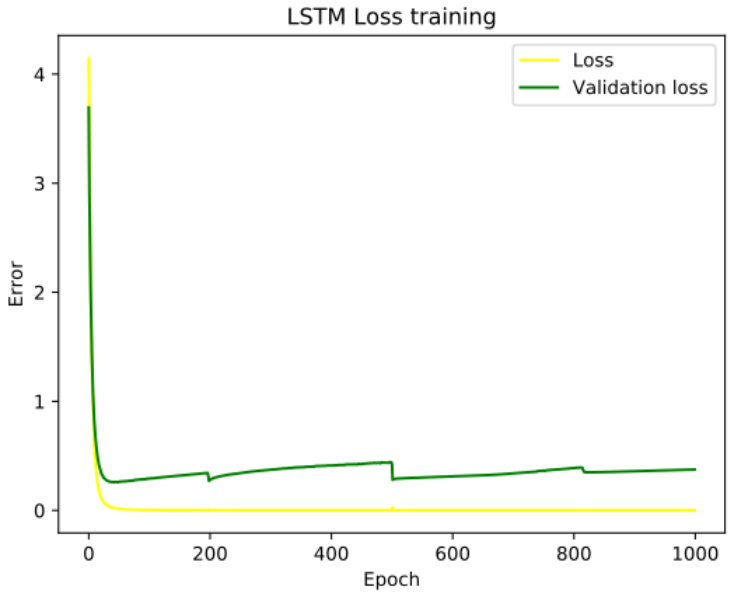


Figura 35: Error empírico (Loss) y de validación (Validation Loss) conforme transcurre el entrenamiento.

Como podemos observar, ambos errores empiezan teniendo un valor alto, el cual desciende rápidamente. Se ha comprobado que ambos se estabilizan cuando la red se encuentra en el epoch 40. De esta manera, escogeremos este valor como número de epochs a entrenar.

En la práctica, para escoger un valor adecuado de epoch, haremos uso de las funcionalidades de Keras, el paquete escogido para hacer funcionar la red LSTM. En este, podemos configurar un "Observador", el cual puede detener el entrenamiento al observar que no hay mejoría (se ha estabilizado), quedándonos con los pesos del epoch de menor error. Esta funcionalidad se ha añadido mediante la siguiente línea:

```
stop_callback=EarlyStopping(monitor='loss',min_delta=0.01,patience=5,verbose=1,mode='min',
    restore_best_weights=True)
history=model.fit( ... , validation_split=0.2, callbacks=[stop_callback])
```

Este "Observador" ha validado la elección hecha, parando en el epoch 41.

Con respecto al método de validación, este ha sido implementado mediante el parámetro "validation_split" que se puede observar en el código arriba. Su valor representa el porcentaje de los datos dados que se usarán como datos de validación. Se ha puesto 0.2, luego el 20 % de los datos se usarán para validar, y el 80 % para entrenar.

5.2.2. Uso de coletilla

Tras elegir el número de epoch, queremos comprobar si la adición de las llamadas "coletillas", ya mencionadas, beneficia o perjudica al tiempo de ejecución de una frase. Para ello, generaremos frases con nuestro modelo LSTM, configurado incluyendo este último cambio de epoch. Estas frases serán medidas, además de medir estas mismas pero añadiendo la coletilla al principio y al final. De esta forma, podemos ver **únicamente** la influencia de estas palabras sobre una frase.

El resultado es el que se muestra en la figura 36.

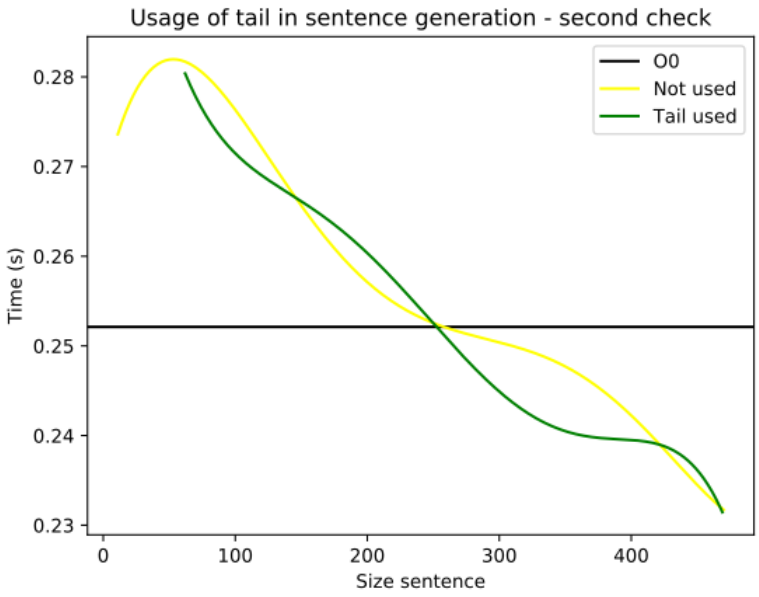


Figura 36: Tiempos de frases con (Tail used) y sin coletilla (Not used).

Como podemos observar, ambos son similares con respecto a tiempos. Ya que su resultado no es concluyente, nos hemos decantado por no incluirla, al no ver mejoría alguna.

Por otro lado, es importante comentar que durante la generación de texto, **generábamos frases con la coletilla como inicio**. Sin embargo, **esas frases no han sido tomadas en cuenta aquí**, ya que nuestro objetivo era comparar frases con y sin coletilla. Si usásemos esas frases, no estaríamos analizando el uso de las coletillas, si no **la inclusión de coletilla como parte de las semillas iniciales**. Este estudio queda como trabajo futuro a estudiar junto con el estudio profundizado de estas semillas, donde sería un factor a tomar en cuenta.

5.2.3. LSTM vs BiLSTM

Bien es sabido que existen múltiples alternativas a la clásica arquitectura de red neuronal recurrente. Nos decantamos por la LSTM, pero existen otras como la ya mencionada BiLSTM, variación bidireccional de LSTM, u otra llamada GRU, la cual varía con respecto de LSTM en cuanto al número y funcionalidad de las puertas por célula. Con el fin de conocer qué arquitectura es mejor para esta finalidad, se desea realizar una comparación entre ellas.

Las arquitecturas que se han mencionado son solo algunas del total existente. Debido a las limitaciones temporales impuestas, no podemos probar con todas. Por esta razón, nos decantamos por probar con la BiLSTM, debido a su mayor similitud con respecto a la LSTM original.

Para compararlos, como anteriormente, se realizará un entrenamiento con ambas arquitecturas. Ambos parten de la codificación Word2Vec, con las decisiones anteriormente tomadas. **En el caso de LSTM, sabemos el número óptimo de epochs que la red debería de entrenar. Con BiLSTM no es así, así que el experimento de los epochs se ha repetido, quedando como resultado una cifra similar: 37 epochs**. Tras entrenar ambos modelos, se generan frases sobre las mismas semillas y se miden los tiempos resultantes.

El resultado aparece en la figura 37.

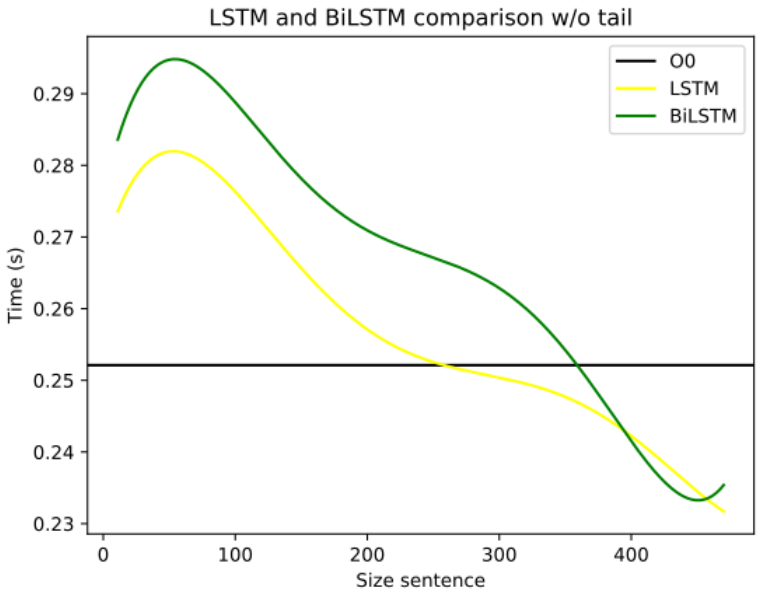


Figura 37: Comparativa de los tiempos obtenidos con LSTM y BiLSTM.

Tras estudiar el resultado obtenido, podemos concluir que se continúa usando la red LSTM, ya que el resultado de la BiLSTM que se muestra es prácticamente igual que el conseguido con la otra red, pero con un desplazamiento superior en el eje Y, en otras palabras, un mayor tiempo.

5.2.4. Comparativa del rendimiento de distintas semillas

Como se dijo previamente, la elección de una semilla adecuada es de suma importancia. La elección hecha, escoger 8 semillas aleatoriamente de entre el 30 % de las mejores frases del corpus, es totalmente adecuada, pero se quería observar el impacto de cambiar estas. De este modo, se han generado frases con distintas semillas. Concretamente, las semillas escogidas son:

- Las 8 mejores frases del corpus.
- Las semillas que hemos usado durante todo el experimento, añadiéndoles la coletilla inicial, y por otro lado, añadir la coletilla final al terminar de generar cada frase (como se comentaba al final del apartado 5.2.2).
- La secuencia de transformaciones correspondiente a O3.

La figura resultante, incluyendo las semillas usadas durante todo el experimento, es la 38, donde "Old seeds" corresponde a las últimas mencionadas, "8 best" a las 8 mejores, "Tail used" en las que se usa la coletilla durante la generación, y "O3" las generadas a partir de O3.

Se aprecia en la figura como claramente el resultado varía de manera sustancial. De aquí podemos sacar varias conclusiones:

- Las frases generadas a partir de las 8 mejores frases del corpus no tienen el mejor resultado. De hecho, es todo lo contrario. Hasta el tamaño 300 de frase (eje X) es la que peor resultado produce, produciendo una mejora sustancial al final. De aquí podemos concluir que, no por tener mejor tiempo significa que sea mejor semilla.

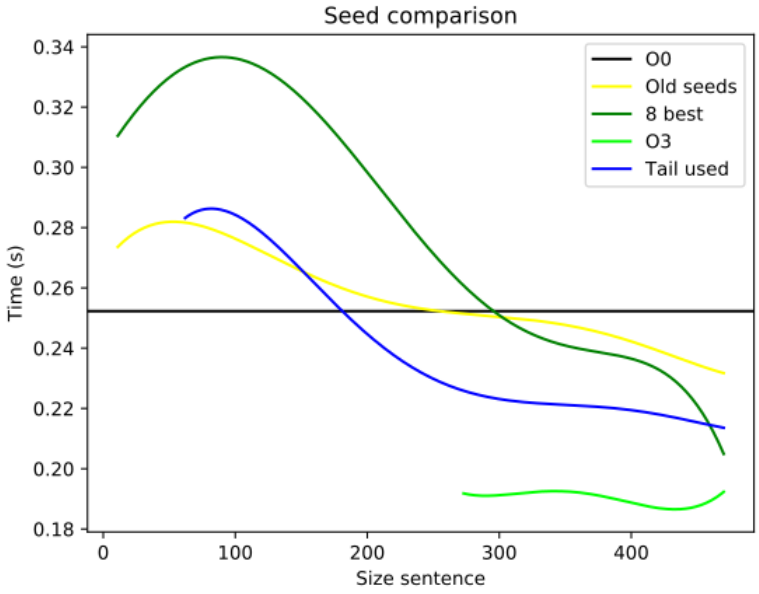


Figura 38: Comparativa de los tiempos obtenidos con frases generadas con distintas semillas y el mismo modelo LSTM.

- La inclusión de coletilla inicial a la hora de generar frases parece producir mejoría sustancial con respecto de no usarla. A partir del valor 250, se aprecia como ambas se encuentran claramente una por encima de la otra, valor que coincide con la longitud de las secuencias ofrecidas por Clang, OX.
- La mejor semilla de las propuestas es O3 según la figura 38, dando un tiempo similar al de la secuencia misma, que ronda alrededor de "0.19" segundos.

  Por otro lado, si observamos la siguiente figura, 39, donde se representa el mínimo tiempo alcanzado para cada longitud de frase (para cada punto del eje X), el resultado parece distinto con respecto del anterior.

  Las frases conseguidas de menor tiempo se encontrarían a partir, de nuevo, de longitud 250, estando los resultados de todas las semillas cercanas uno con respecto de los otros. En este caso, el mejor resultado estaría en la semilla de las 8 mejores frases del corpus.

  Podemos concluir que, aunque la "tendencia general" de una semilla sea de tiempo alto, puede que el tiempo mínimo que este consigue, sea mejor.

5.3. LSTM vs Genético

Tras haber comparado varias arquitecturas de redes neuronales, se pretende hacer una comparación adicional. Esta vez, en vez de comparar con otra red neuronal, se compara con el algoritmo genético. Esta comparación tiene como motivo el evaluar cuán útil es el uso de las redes neuronales, más complejas, frente a un algoritmo que carece de entrenamiento u aprendizaje, uno más clásico.

Para ello, el algoritmo genético, por un lado, generará frases usando el corpus completo, del que también se ha tenido acceso durante la fase de entrenamiento en las redes. Por otro lado, las frases usadas para comparar serán las ya generadas en la anterior comparación. Concretamente, se usarán las frases de la semilla O3. Además, como es evidente, ambos algoritmos no pueden usar las mismas configuraciones al ser de distinto funcionamiento.

Los hiperparámetros usados para el algoritmo genético, los cuales ya aparecen en 4.9, son los siguientes:

- Porcentaje máximo con respecto de la población total que un individuo puede aparecer (parámetro rate_max_solutions): 0.3, es decir, 30 %.
- Probabilidad de mutación: 0.1, 10 %.
- Probabilidad de cruce: 0.2, 20 %.

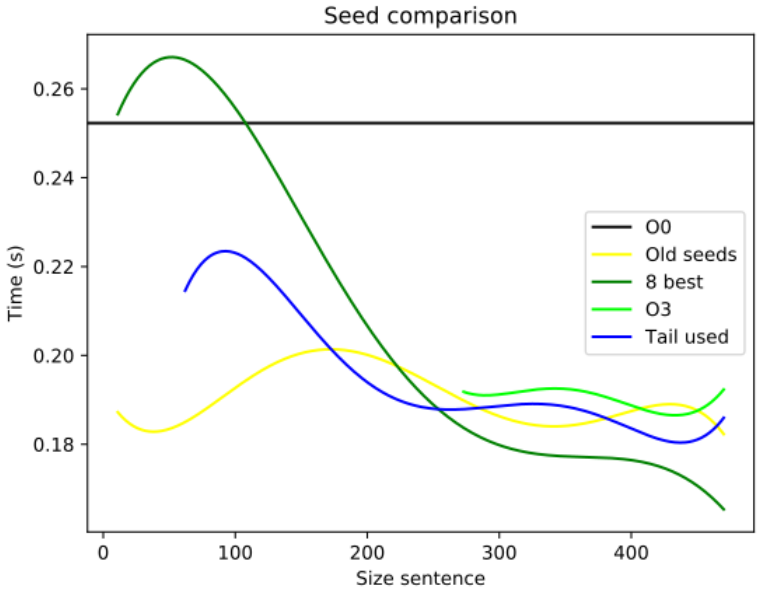


Figura 39: Segunda comparativa de los tiempos obtenidos con frases generadas con distintas semillas y el mismo modelo LSTM.

- Operador de mutación: Mutación polinomial para enteros.
- Operador de cruce: Cruce SBX para enteros.
- Selección: Ruleta.
- Criterio de finalización: Que el algoritmo haga 10 iteraciones.
- Población: Tantas como frases en el corpus, asegurándonos de que sea par (quitar una frase en caso de impar).

Finalmente, la comparativa de ambos se puede observar en la figura 40 a continuación.

La conclusión que podemos sacar es que el algoritmo genético, en las condiciones actuales, y con las semillas que hemos usado, produce un mejor tiempo que con la red LSTM, aunque no por mucha diferencia.

Podemos apreciar, entre los valores 400 y 450 del eje X, que el resultado de O3 llega a incluso superar todos los flags. Si incluyéramos el resto de semillas tal y como en 39, veríamos como **todas las semillas previamente generadas llegan a superar a todas las secuencias ofrecidas por Clang, los OX**. De igual forma, **el algoritmo genético también las supera**.

Si usásemos las semillas generadas por el algoritmo genético como entrada para el LLVM, cabe la posibilidad de que consigamos un tiempo similar o incluso mejor. De este modo, queda aún margen de mejora.

5.4. Comparativa con el corpus inicial, la aleatoriedad, y las secuencias originales

Finalmente, se plantea una serie de comparativas más a hacer. Su finalidad es comprobar la calidad del resultado obtenido frente a, no algoritmos, si no otros datos con los que se considera de alta utilidad el medirse.

El primero de estos datos es el corpus. Observando la distancia con respecto de él, se puede observar si se ha conseguido un mejor resultado con respecto de los datos de los que se partió. A continuación, se encuentra la aleatoriedad. Si nuestro sistema queda en peor lugar que la aleatoriedad, cabría la posibilidad de que nuestro enfoque no sea el deseado. En tal caso, podríamos plantear alternativas. Finalmente, nos compararíamos con nuestra "meta" u objetivo, las secuencias de Clang, para observar si conseguimos o no un nivel de optimización similar (o no) al ya disponible en LLVM. Este es uno de los objetivos iniciales del proyecto.

Para la generación de frases aleatorias, simplemente se utilizan las librerías "random" de Python para generar tantos números como tan larga sea la frase de mayor longitud. Estos números irían de 0 a 87, correspondiendo a las palabras de nuestro vocabulario.

El resultado de todo ello es el que se muestra en la figura 41.

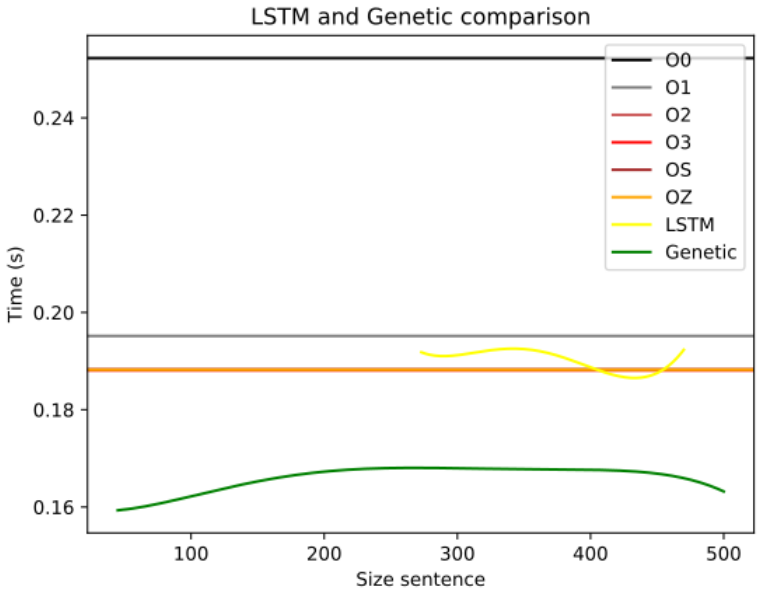


Figura 40: Comparación entre el algoritmo genético, LSTM con la semilla O3 y todas las secuencias de Clang.

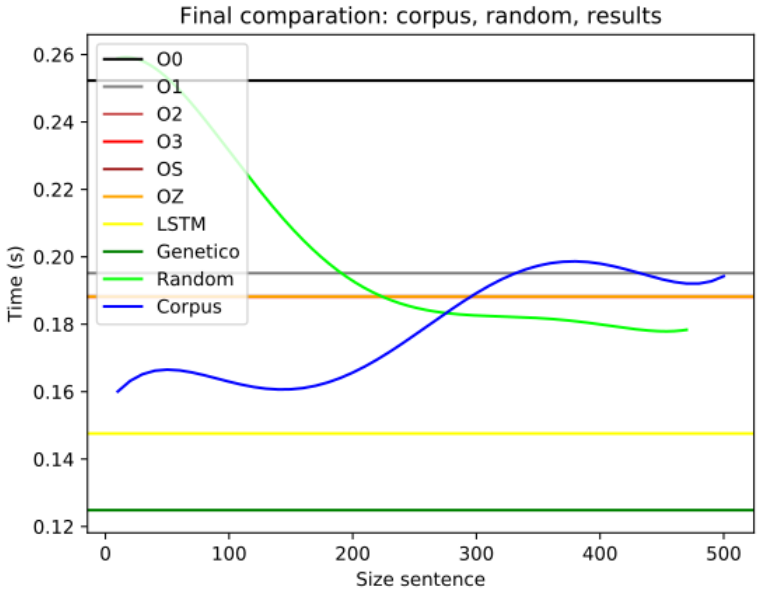


Figura 41: Comparativa entre los tiempos del corpus inicial, frases generadas aleatoriamente, y los resultados conseguidos.

Para la realización de esta gráfica, se ha escogido el mejor resultado generado tanto por la red LSTM, como el mejor del algoritmo genético.

Como se puede ver en la parte inferior de la figura, **ambos resultados de genético y LSTM superan con creces a las secuencias ofrecidas de Clang, al corpus, y a las frases generadas aleatoriamente**.

De este modo, se puede concluir que, en vista de las decisiones tomadas, **es factible la generación de secuencias de transformaciones usando algoritmos de procesado de lenguaje natural y Deep Learning**. Pese a que el genético obtiene una mejor solución hasta el momento, este tiene un coste computacional mucho más elevado que la alternativa planteada en este proyecto. Queda, pues, aún margen de mejora, pero podemos afirmar su factibilidad exitosamente.

6. Conclusión

Tomando una perspectiva general, podemos describir los siguientes puntos de este proyecto:

- Generación de un corpus inicial a partir de las secuencias de transformaciones de LLVM.
- Se ha establecido un método de comparación de secuencias de transformaciones, las mediciones de tiempo.
- Configuración de un Benchmark necesario para las comparaciones.
- Se han obtenido los tiempos de las secuencias de transformaciones de LLVM, que sirven como referencia comparativa.
- Generación de una codificación "Word Vector" basada en el algoritmo Word2Vec, de procesamiento de lenguaje natural.
- Se han configurado este algoritmo, comparando configuraciones y tomando criterios que mejoran la codificación resultante.
- Se ha comprobado la eficacia del uso de esta codificación.
- Generación de frases mediante el uso de una red neuronal LSTM.
- Comparativa de distintas frases como origen desde el que generar otras (semillas).
- Comparativa de varias arquitecturas de redes, así como un algoritmo de enfoque clásico, el Genético.
- Comparativa de las secuencias generadas con respecto de la frases aleatorias, el corpus inicial del que se parte y las secuencias de transformaciones LLVM, usadas como referencia para medirnos.

Finalmente, podemos concluir de esta última comparativa que, tras las decisiones tomadas justificadamente, **podemos afirmar la factibilidad de la generación de secuencias de transformaciones usando procesado de lenguaje natural (NLP) y Deep Learning**, pese al margen de mejora que cabe.

7. Trabajo futuro

A raíz del alcance y la limitación temporal del proyecto, se plantea una serie de elementos a tratar en un futuro, si se desea continuar con este estudio. Concretamente:

- Uso de otras arquitecturas de redes neuronales, un ejemplo podría ser la red neuronal GRU, u otros algoritmos con los que generar frases para, posteriormente, realizar una comparativa como las ya hechas.
- Estudio de las transformaciones LLVM, con el fin de realizar un gran volumen de analogías y similitudes, con los que poder evaluar la codificación resultante desde una perspectiva NLP, en el caso de Word2Vec. Este gran volumen tendría que ser obtenido mediante anotación manual.
- Búsqueda de una configuración óptima usando el gran volumen de analogías previamente generado.
- Comparativa de la codificación usada, word vector (obtenida de aplicar Word2Vec), con una configuración óptima, con otras existentes.
- Generación de otras frases, con la arquitectura usada en el proyecto, partiendo de semillas de menor tiempo a las usadas. Estas han sido conseguidas al generar nuevas frases, las cuales hemos visto en el apartado de discusión.
- Estudio de qué características debe tener una frase para ser una semilla eficaz: longitud, tiempo, etc.
- Búsqueda de nuevos criterios o ideas que hagan mejorar o bien los resultados de los modelos usados, LSTM o Word2Vec, o bien los tiempos obtenidos al generar frases. En este proyecto se han planteado algunos, tales como las coletillas, la reducción del tamaño del corpus, etc.
- Comparativa entre distintos métodos de sampling, o método de elección de la siguiente palabra, en el caso de este proyecto, los cuales ya fueron mencionados.
- Estudio de si existe un tamaño óptimo hasta el que debemos hacer crecer las frases (generar frases hasta que tengan un tamaño específico).
- Continuar con el estudio del uso de algoritmos genéticos.

Referencias

[1] Amir Hossein Ashouri. Compiler autotuning using machine learning techniques. *URL https://www.politesi.polimi.it/bitstream/10589/129561/1/ASHOURI_PhD_thesis_2016.pdf*, pages 3–4, 2016.

[2] Mu Li Alexander J. Smola Aston Zhang, Zachary C. Lipton. *Dive into Deep Learning*, volume 1, page 378. Corwin Press, 7 2019. URL https://d2l.ai/d2l-en.pdf, último acceso 8 de febrero de 2021.

[3] Steve Zdancewic at Penn. Llvm and ssa. Disponible en https://flint.cs.yale.edu/cs421/lectureNotes/Spring15/llvm.pdf, 1 2015.

[4] Patrick van der Smagt Ben Kröse. *An introduction to neural networks*, chapter 4. , 8 edition, 1996. URL https://www.infor.uva.es/ teodoro/neuro-intro.pdf.

[5] Antonio Benítez-Hidalgo, Antonio J. Nebro, José García-Nieto, Izaskun Oregi, and Javier Del Ser. jmetalpy: A python framework for multi-objective optimization with metaheuristics. *Swarm and Evolutionary Computation*, page 100598, 2019.

[6] Denny Britz. Recurrent neural networks tutorial, part 3 – backpropagation through time and vanishing gradients. wildml.com, 10 2015. URL http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/7/, última vez accedido 8 de febrero de 2021.

[7] Amy Brown and Greg Wilson. *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*, volume 1, chapter 11. Lulu. com, 2011.

[8] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[9] Z Cui, R Ke, Z Pu, and Y Wang. Stacked bidirectional and unidirectional lstm recurrent neural network for network-wide traffic speed prediction. *arXiv preprint arXiv:1801.02143*, 2018.

[10] Wei Dai and Daniel Berleant. Benchmarking contemporary deep learning hardware and frameworks: A survey of qualitative metrics. In *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)*, pages 148–155. IEEE, 2019.

[11] Juan Carlos de la Torre, Patricia Ruiz, Bernabé Dorronsoro, and Pedro L Galindo. Analyzing the influence of llvm code optimization passes on software performance. In *International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, pages 272–283. Springer, 2018.

[12] Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 297–304, 2010.

[13] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. karpathy.github.io, 5 2015. Disponible en http://karpathy.github.io/2015/05/21/rnn-effectiveness/ , última vez accedido 8 de febrero de 2021.

[14] Chris Lattner. Introduction to the llvm compiler infrastructure. In *Itanium conference and expo*, pages 15, 16, 17, 2006.

[15] Chris Lattner. Introduction to the llvm compiler system. In *Proceedings of International Workshop on Advanced Computing and Analysis Techniques in Physics Research, Erice, Sicily, Italy*, pages 30, 34, 2008.

[16] Chris Lattner et al. The llvm compiler infrastructure. *URL http://llvm. org*, 2010.

[17] Omer Levy and Yoav Goldberg. Dependency-based word embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 302–308, 2014.

[18] Pierre Lison and Andrey Kutuzov. Redefining context windows for word embedding models: An experimental study. *arXiv preprint arXiv:1704.05781*, 2017.

[19] Bruno Cardoso Lopes and Rafael Auler. *Getting started with LLVM core libraries*, chapter 3.2. Packt Publishing Ltd, 2014.

[20] Bruno Cardoso Lopes and Rafael Auler. *Getting started with LLVM core libraries*, chapter 5.5.4. Packt Publishing Ltd, 2014.

[21] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[22] Andriy Mnih and Geoffrey E Hinton. A scalable hierarchical distributed language model. In *Advances in neural information processing systems*, pages 1081–1088, 2009.

[23] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252. Citeseer, 2005.

[24] Chris Nicholson. A beginner's guide to word2vec and neural word embeddings. pathmind.com. URL https://pathmind.com/wiki/word2vec, último acceso 8 de febrero de 2021.

[25] Christopher Olah. Understanding lstm networks. http://colah.github.io, 8 2015. URL http://colah.github.io/posts/2015-08-Understanding-LSTMs/, último acceso 8 de febrero de 2021.

[26] Kevin Patel and Pushpak Bhattacharyya. Towards lower bounds on number of dimensions for word embeddings. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 31–36, 2017.

[27] Mohd Sanad Zaki Rizvi. Demystifying bert: A comprehensive guide to the groundbreaking nlp framework, 9 2019. URL https://www.analyticsvidhya.com/blog/2019/09/demystifying-bert-groundbreaking-nlp-framework/, último acceso 8 de febrero de 2021.

[28] Xin Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.

[29] Thijs Scheepers. tscheepers's word2vec implementation. Disponible en https://github.com/tscheepers/word2vec, 2015.

[30] Noah A Smith and Jason Eisner. Contrastive estimation: Training log-linear models on unlabeled data. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 354–362, 2005.

[31] Jordi Torres. *Deep Learning – Introducción práctica con Keras (SEGUNDA PARTE)*, chapter 7. Lulu. com, 2018.